COVERS FLASH BASIC 8 AND FLASH PROFESSIONAL 8!

Foundation

ActionScript for Flash 8

Kristian Besley
Sham Bhangal
David Powers
with Eric Dolecki

Master ActionScripting the easy way!

Give your Flash movies intelligence

Explore the new features of Flash 8

friendsof ED
DESIGNER TO DESIGNER™
an Apress® company

Foundation ActionScript for Flash 8

Kristian Besley
Sham Bhangal
David Powers
with Eric Dolecki

friendsof ED™
DESIGNER TO DESIGNER™
an Apress® company

Foundation ActionScript for Flash 8

ISBN-13 (pbk): 978-1-59059-618-0

ISBN-10 (pbk): 1-59059-618-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor
Chris Mills

Technical Reviewers
Kristian Besley and David Powers

Editorial Board
Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade

Project Manager
Denise Santoro Lincoln

Copy Edit Manager
Nicole LeClerc

Copy Editors
Ami Knox and Nicole LeClerc

Assistant Production Director
Kari Brooks-Copony

Production Editor
Kelly Winquist

Compositor
Dina Quan

Artist
Kinetic Publishing Services, LLC

Proofreader
Linda Seifert

Indexer
Julie Grady

Cover Image Designer
Corné van Dooren

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

CONTENTS AT A GLANCE

CONTENTS

ABOUT THE AUTHORS

Image courtesy of Simon James at `www.thefresh.co.uk`

Kristian Besley is a Flash/web developer working in education and specializing in interactivity and dynamically driven content using ASP.NET and PHP. In his spare time, he is also a lecturer in multimedia at the higher education level.

Kristian has written a number of friends of ED books, such as the *Foundation Flash* series (including the recently published *Foundation Flash 8*), *Flash MX Video*, and *Learn Programming with Flash MX*. He was a contributor to the *Flash Math Creativity* books, *Flash MX 2004 Games Most Wanted*, *Flash MX Video Creativity*, and countless others. He also writes for *Computer Arts* magazine and has produced freelance work for numerous clients, including the BBC.

Kristian currently resides in Swansea, Wales, the city of his birth. He is a fluent Welsh speaker and is the creator of the first-ever Welsh translation search plug-in for Firefox and Mozilla (available from `http://mycroft.mozdev.org`).

Sham Bhangal has written on new media for friends of ED since the imprint's inception. In that time, he has been involved in the writing, production, and specification of just under 20 books.

Sham has considerable working experience with Macromedia and Adobe products, with a focus on web design and motion graphics. Creating books that tell other people about his favorite subjects is probably the best job he has had (ignoring the long hours, aggressive deadlines, lost manuscripts, and occasional wiped hard drives). If he was doing something else, he'd probably be losing sleep thinking about writing anyway.

Sham currently lives in the north of England with his longtime partner, Karen.

David Powers is a professional writer who has been involved in electronic media for more than 30 years, first with BBC radio and television, and more recently with the Internet. This is his sixth book for Apress/friends of ED on programming for the Web. Among his previous titles are the highly successful *Foundation PHP 5 for Flash* (friends of ED, ISBN: 1-59059-466-5) and *Foundation PHP for Dreamweaver 8* (friends of ED, ISBN: 1-59059-569-6). David's other main area of expertise is Japan. He was a BBC correspondent in Tokyo during the late 1980s and early 1990s, and later was Editor, BBC Japanese TV. He has also translated several plays from Japanese.

ABOUT THE COVER IMAGE DESIGNER

Corné van Dooren designed the front cover image for this book. Having been given a brief by friends of ED to create a new design for the Foundation series, he was inspired to create this new setup combining technology and organic forms.

With a colorful background as an avid cartoonist, Corné discovered the infinite world of multimedia at the age of 17—a journey of discovery that hasn't stopped since. His mantra has always been "The only limit to multimedia is the imagination," a philosophy that is keeping him moving forward constantly.

After enjoying success after success over the past years—working for many international clients, as well as being featured in multimedia magazines, testing software, and working on many other friends of ED books—Corné decided it was time to take another step in his career by launching his own company, Project 79, in March 2005.

You can see more of Corné's work and contact him through `www.cornevandooren.com` or `www.project79.com`. If you like his work, be sure to check out his chapter in *New Masters of Photoshop: Volume 2* (friends of ED, ISBN: 1-59059-315-4).

INTRODUCTION

Welcome to *Foundation ActionScript for Flash 8*, the fourth edition of this legendary ActionScript book.

ActionScript is, quite simply, the driving force behind Flash applications, allowing you to go beyond simple tweened animations and give your movies intelligence, power, and class! The current version of ActionScript in Flash 8, 2.0, is a fully featured, very powerful programming language.

But ActionScript is that scary code stuff that programmers do, right? Wrong. ActionScript adds power and potential to your design work. It's not going to turn you into a reclusive nerd who speaks in 1s and 0s, and who only comes out after dark. It's going to turn you into someone who finally has the power to achieve his or her web design goals, rather than being hemmed in by frustrating limitations.

And Flash 8 has a treasure trove of new features for you to play with. It has amazing new design features such as filters and blend modes, features such as bitmap caching to enhance the speed of your movies, exciting new video capabilities, a great new BitmapData API for manipulating images on the fly, and much more.

If you know nothing or little about ActionScript, this book will provide you with a real foundation of knowledge from which you can build some awe-inspiring structures. You'll learn all the important stuff you'll need to make that giant leap toward becoming an ActionScript guru.

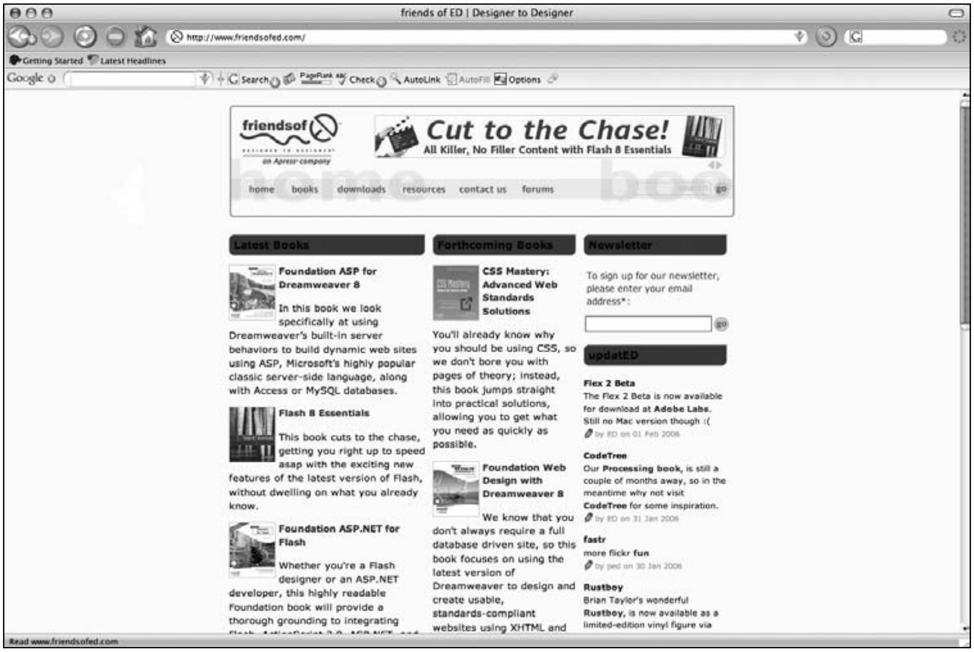
What you need to know

You've picked up this book, so we guess that you have the Flash basics under your belt. You'll probably have built some basic timeline-based movies with tweens and so on, and you may have read an introductory Flash book such as friends of ED's acclaimed *Foundation Flash 8*. If you haven't, we do recommend looking at it; you can find it at `www.friendsofed.com`.

If you don't have a copy of Flash 8 yet, you can download a fully functional 30-day free trial from `www.macromedia.com`. You can use either the Basic or Professional edition of Flash with this book, but we highly recommend going for Professional, as it features even more amazing functionality than the Basic edition!

FLAs for download

There's a wealth of code and support files available for this book. They're organized by chapter at the *Foundation ActionScript for Flash 8* page at `www.friendsofed.com`. Look under the book option on the site's main navigation to find it, and feel free to look around the site in general!



Foundation ASP for Dreamweaver 8

In this book we look specifically at using Dreamweaver's built-in server behaviors to build dynamic web sites using ASP, Microsoft's highly popular classic server-side language, along with Access or MySQL databases.

Flash 8 Essentials

This book cuts to the chase, getting you right up to speed asap with the exciting new features of the latest version of Flash, without dwelling on what you already know.

Foundation ASP.NET for Flash

Whether you're a Flash designer or an ASP.NET developer, this highly readable Foundation book will provide a thorough grounding to integrating

CSS Mastery: Advanced Web Standards Solutions

You'll already know why you should be using CSS, so we don't bore you with pages of theory; instead, this book jumps straight into practical solutions, allowing you to get what you need as quickly as possible.

Foundation Web Design with Dreamweaver 8

We know that you don't always require a full database driven site, so this book focuses on using the latest version of Dreamweaver to design and create usable, standards-compliant websites using XHTML and

To sign up for our newsletter, please enter your email address*:

Flex 2 Beta
The Flex 2 Beta is now available for download at **Adobe Labs**. Still no Mac version though :(
by ED on 01 Feb 2006

CodeTree
Our **Processing book**, is still a couple of months away, so in the meantime why not visit **CodeTree** for some inspiration.
by ED on 31 Jan 2006

fastr
more flickr **fun**
by ped on 30 Jan 2006

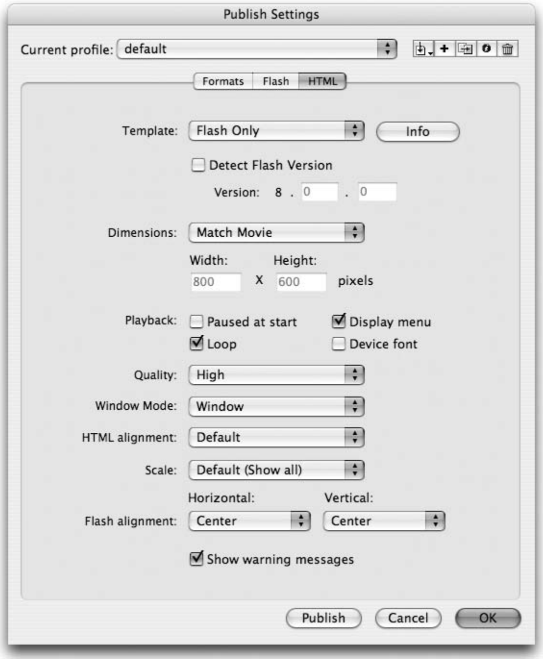
Rustboy
Brian Taylor's wonderful **Rustboy**, is now available as a limited-edition vinyl figure via

The case study: Futuremedia

Throughout the course of this book you'll create a website called Futuremedia from scratch. You can access a fully functioning version of the website you'll be building as you progress through this book at the URL you'll find on the downloads page (or you can go to `www.futuremedia.org.uk` for the latest incarnation).

Centering the Futuremedia site in the browser

When you publish the Futuremedia site, you should use the following settings in the File ➤ Publish Settings ➤ HTML tab:



Publish Settings

Current profile: default

Formats | Flash | HTML

Template: Flash Only [Info]

□ Detect Flash Version

Version: 8 . 0 . 0

Dimensions: Match Movie

Width: 800 X Height: 600 pixels

Playback: □ Paused at start ☑ Display menu
☑ Loop □ Device font

Quality: High

Window Mode: Window

HTML alignment: Default

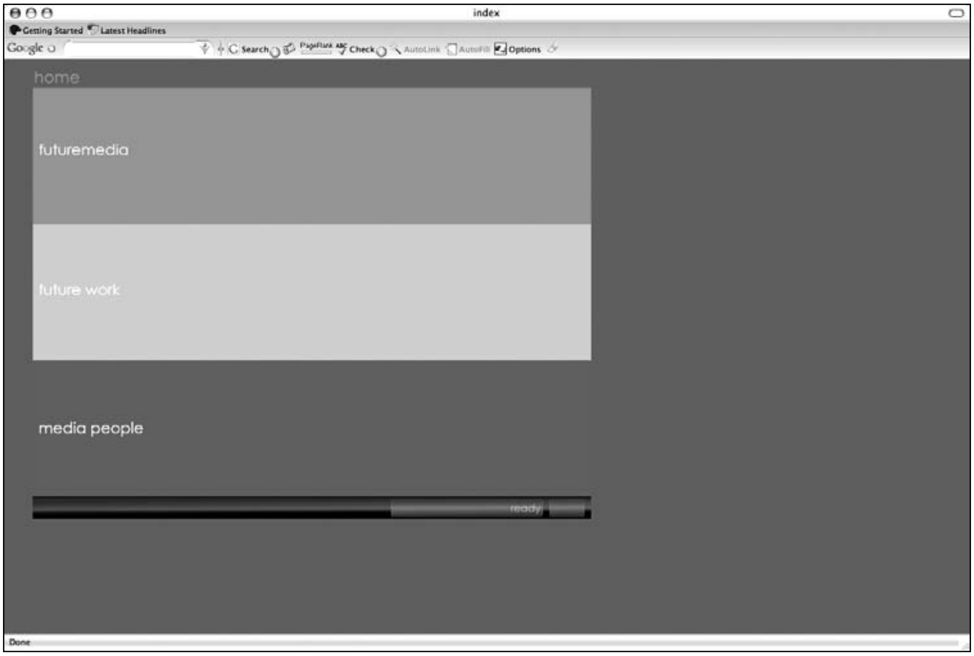
Scale: Default (Show all)

Flash alignment: Horizontal: Center Vertical: Center

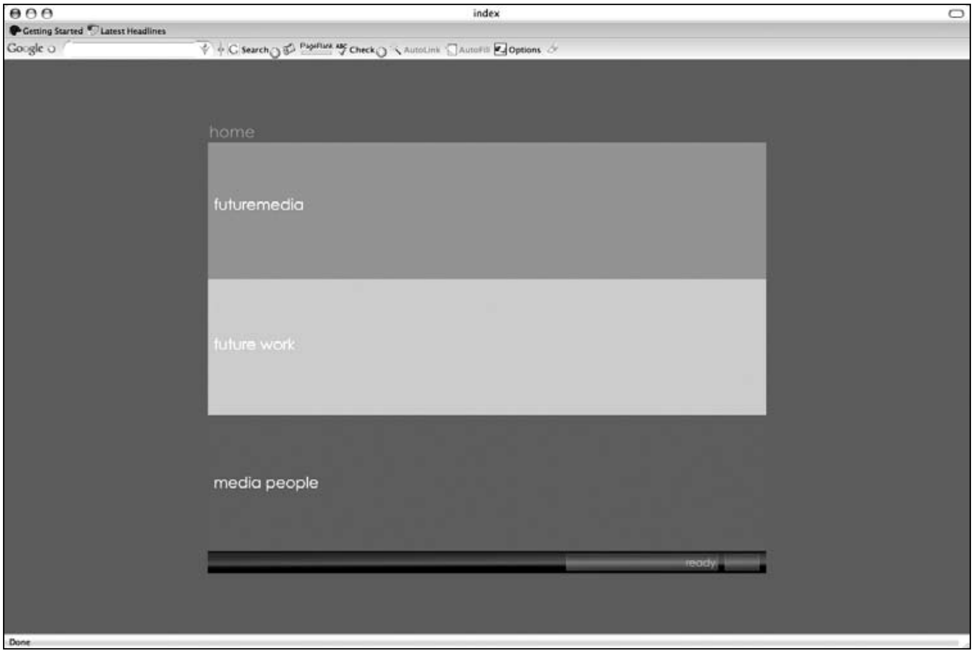
☑ Show warning messages

[Publish] [Cancel] [OK]

With these settings, you'll see something like this in the browser (press F12 to publish the site and view it in your browser):



That's fine, but most professional sites center the Flash site in the browser, so it looks something like this instead:



There's no direct way of achieving this in Flash—you have to edit the HTML. To do this, find the HTML file created by Flash (it will be in the same folder as the FLA), and open it in a text editor such as Notepad. You'll see something like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns=http://www.w3.org/1999/xhtml xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html;➥
  charset=iso-8859-1" />
<title>index</title>
</head>
<body bgcolor="#666666">
<!--urls used in the movie-->
<!--text used in the movie-->
<!--
  futuremedia
  future work
  media people
  loading:
  this is a skip-intro free site
-->
```

```
<object classid="clsid:d27cdb6e-ae6d-➥
  11cf-96b8-444553540000" codebase=➥
  "http://fpdownload.macromedia.com/pub/➥
  shockwave/cabs/flash/swflash.cab#➥
  version=8,0,0,0" width="800" height=➥
  "600" id="index" align="middle">
<param name="allowScriptAccess" value="sameDomain" />
<param name="movie" value="index.swf" />
<param name="quality" value="high" />
<param name="bgcolor" value="#666666" />
<embed src="index.swf" quality="high"➥
  bgcolor="#666666" width="800" height=➥
  "600" name="index" align="middle"➥
  allowScriptAccess="sameDomain" type=➥
  "application/x-shockwave-flash"➥
  pluginspage="http://www.macromedia.com/➥
  go/getflashplayer" />
</object>
</body>
</html>
```

This is XHTML, so you should really play ball and use CSS and `<div>` and `<span>`, and no HTML tables or table horizontal and vertical centering (not least because vertical centering of a table doesn't work in XHTML!). Add the following lines to create a CSS-based centered-in-browser page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns=http://www.w3.org/1999/xhtml xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html;➥
  charset=iso-8859-1" />
<title>index</title>
<style type="text/css">
<!--
body
{
 margin: 0px;
 background-color:#666666;
}
#centercontent
{
 text-align: center;
 margin-top: -300px;
 margin-left: -400px;
 position: absolute;
 top: 50%;
 left: 50%;
}
```

```
-->
</style>
</head>
<body>
<!--urls used in the movie-->
<!--text used in the movie-->
<!--
  futuremedia
  future work
  media people
  loading:
  this is a skip-intro free site
-->
<div id="centercontent">
<object classid="clsid:d27cdb6e-ae6d-➥
  11cf-96b8-444553540000" codebase=➥
  "http://fpdownload.macromedia.com/pub/➥
  shockwave/cabs/flash/swflash.cab#➥
  version=8,0,0,0" width="800" height=➥
  "600" id="index" align="middle">
<param name="allowScriptAccess" value="sameDomain" />
<param name="movie" value="index.swf" />
<param name="quality" value="high" />
<param name="bgcolor" value="#666666" />
<embed src="index.swf" quality="high"➥
  bgcolor="#666666" width="800" height=➥
  "600" name="index" align="middle"➥
  allowScriptAccess="sameDomain" type=➥
  "application/x-shockwave-flash"➥
  pluginspage="http://www.macromedia.com/➥
  go/getflashplayer" />
</object>
</div>
</body>
</html>
```

Note also that the deprecated `bgcolor` attribute has been removed from the `<body>` element and replaced with a nice shiny new standards-compliant CSS rule.

Layout conventions

To keep this book as clear and easy to follow as possible, the following text conventions are used throughout.

Important words or concepts are normally highlighted on the first appearance in **bold type**.

Code is presented in `fixed-width font`.

New or changed code is normally presented in **`bold fixed-width font`**.

Pseudo-code and variable input are written in *`italic fixed-width font`*.

Menu commands are written in the form Menu ➤ Submenu ➤ Submenu.

Where we want to draw your attention to something, we've highlighted it like this:

> *Ahem, don't say we didn't warn you.*

Sometimes code won't fit on a single line in a book. Where this happens, we use an arrow like this: ➥.

```
This is a very, very long section of code that should be written ➥
all on the same line without a break.
```

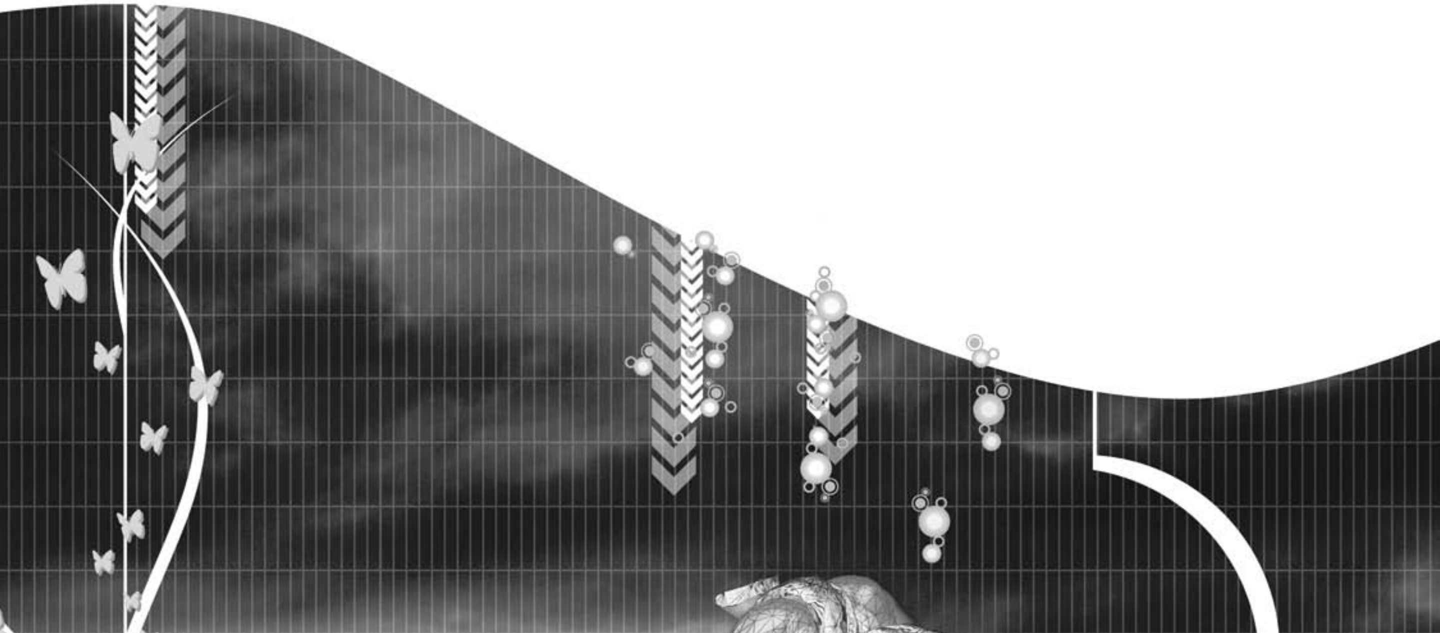
PCs and Macs

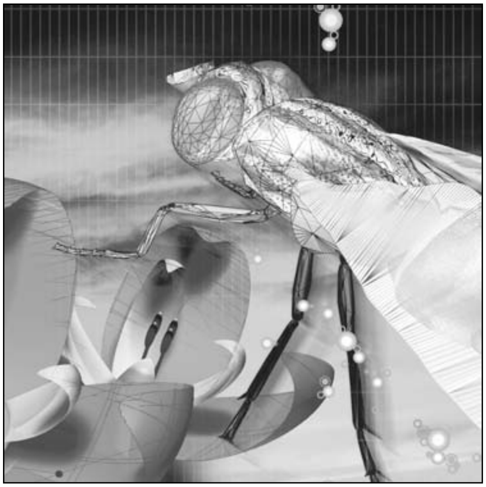
To make sure this book is as useful to you as possible, we've tried to avoid making too many assumptions about whether you're running Flash on a PC or a Mac. However, when it comes to mouse clicks and modifier buttons, we can't generalize. There's no two ways about it: they're different!

When we use the term "click," we mean left-click on the PC or simply click on the Mac. On the other hand, a right-click on the PC corresponds to holding down the Ctrl button and clicking on the Mac. This is abbreviated as Ctrl-click.

Another important key combination on the PC is when you hold down the Ctrl key while you press another key (or click something). This sort of thing is abbreviated as Ctrl-click. The Mac equivalent is to hold down the Cmd key (also known as the "apple" or "flower" key) instead, so you'll normally see this written out in the form Ctrl+C/Cmd+C, or Ctrl+V/Cmd+V, for example.

OK, now that we've taken care of the preliminaries, let's get to work!





Chapter 1

INTERACTIVE FLASH

What we'll cover in this chapter:

- Introducing ActionScript and the Actions panel
- Using actions to give commands to a Flash movie
- Targeting actions at different things in a movie
- Listening to Flash and handling Flash events
- Writing callbacks so that real-time events can trigger actions
- Using variables as containers for storing information

So, what's this ActionScript business all about? You may well have done loads of great stuff with Flash already, but you still keep hearing people say you've barely scratched the surface because you haven't learned to **program** your movies. On the other hand, programming is supposed to be a strange and highly complex business, practiced by myopic young people with pale skin and peculiar taste in music. Programming is for hackers, not artists. That's a path in life many would probably rather avoid.

Fortunately, there's some good news. Programming is a bit like mountaineering: there are dedicated full-timers who might spend every waking minute preparing to tackle the north face of Eiger, even if that means their day-to-day lives come to resemble some kind of hideous training regimen. However, there are plenty more folks who'll just take a few weeks out from "normal life" to join a trek up Mount Kilimanjaro and get back home fit, tanned, and with a whole new perspective on things—not to mention a great big wad of stunning photos!

We're not going to suggest that learning ActionScript is somehow going to get you fit and tanned. But it can help you to race ahead with making rich, powerful, truly fantastic Flash movies that will leave most non-ActionScripting Flash-meisters wondering how to even begin creating such things. So, to get back to the point, what's this ActionScript business all about? Well, as far too many irritating game show hosts have reminded us over the years, "The clue is in the question." There are two words to consider:

Action

Script

We all know what these words mean: an **action** is something you do, and a **script** is something actors read so that they know what to say and do when they go on stage or in front of a camera. Well, that's really all there is to it. ActionScript is like a script that you give to a Flash movie, so that you can tell it what to do and when to do it—that is, you give it **actions** to perform.

> *The analogy between ActionScript and film scripts is more than just a metaphor. When writing ActionScript, I often like to assume that I'm writing scripts for human actors, and before I write code, I figure out how I would say what I want to do to a real actor before implementing it in ActionScript. This is actually a very good test—if you find you aren't able to find the words needed, it's probably because you don't really understand your problem, and it's time to turn away from the machine and work it out on paper first.*
>
> *—Sham Bhangal*

Giving your movies instructions

Actions are the basic building blocks of ActionScript. It's sometimes helpful to think of them as instructions you give to the movie, or even better, as **commands**. You tell the movie to stop. You tell it to start again. You tell it to go to such and such a frame, and start playing from there. You probably get the idea. Whenever you want to tell your movie to do something that it wouldn't normally do anyway, you can do so by giving it an action.

Since this book is about ActionScript, we'll stick to using the word "actions." If you mention actions to programmers who don't use Flash, though, they may look at you blankly for a moment before saying, "Ah, you mean commands!" In our view, it's well worth keeping the idea that actions are the same thing as commands at the back of your mind.

As you know if you've sneaked a look already, there are stacks of different actions available in Flash, which makes it (a) possible to do all sorts of weird and wonderful things to your movies, and (b) pretty hard to know where to start. Don't worry, though, we have a cunning plan! You'll start off by looking at some simple actions that let you start and stop a movie's main timeline, and then you'll move on to look at jumping about from one frame to another. Nothing too tricky to begin with.

Before you start writing actions, though, we need to make an important introduction. The **Actions panel** will be your partner in crime throughout this chapter, the rest of the book, and your future career as an ActionScript hero. Let's check it out.

If you own the Professional version of Flash, you have the ability to open a full-screen ActionScript editor by choosing File ➤ New *and then selecting* ActionScript File *from the* General *tab of the* New Document *window that will appear. This editor is actually very similar to the normal* Actions *panel. The major difference with the full-screen editor is it's just that—there's no stage and no timeline. The full-screen editor is designed for writing a stand-alone ActionScript file, something you'll have the option to use as you build your website. Just because this is a beginner book doesn't mean we're not going to fully cover the Professional version of Flash as well as the Basic edition, so don't worry, we haven't forgotten you!*

Working with the Actions panel

The Actions panel is where you'll be writing all your ActionScript, so it will pay for you to know your way around it before you make a proper start. Before you do, though, let's make sure we're all on the same page. Open a new Flash document and select Window ➤ Workspace Layout ➤ Default, and the interface should go back to the out-of-the-box state.

If you ever get to a point at which you have many panels open and a generally cluttered screen, Window ➤ Workspace Layout ➤ Default *is a good option to select—you don't lose your work in progress by doing it. It's also a good idea to select this option before you start any of the step-by-step tutorials.*

First, you need to open up the Actions panel. As with any panel, there are several ways to do this, but for now we'll go for the Window ➤ Actions menu selection as the cross-platform-friendly option.

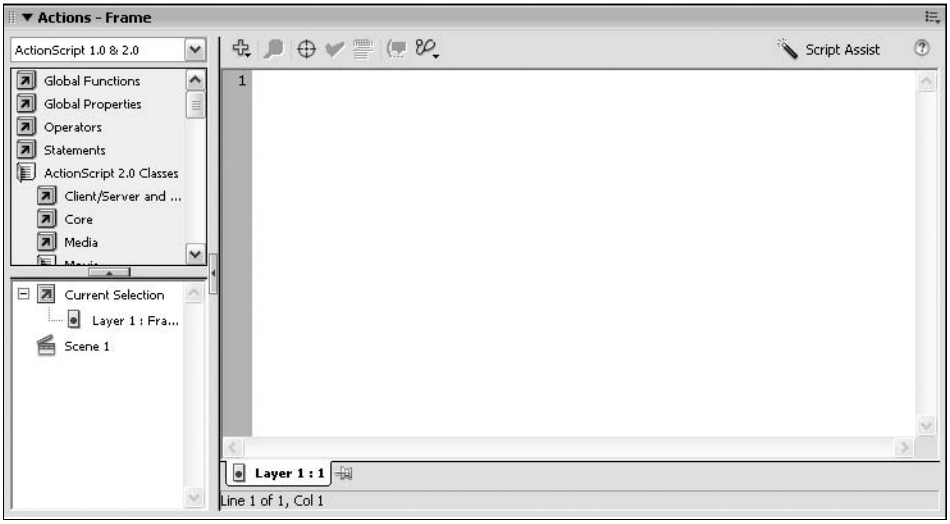
You can also open (and close) the Actions panel with the F9 keyboard shortcut.

Here's a can of worms . . .

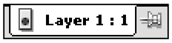
If you're using a Mac with Exposé enabled with the default settings, you'll soon discover that pressing F9 triggers Exposé and doesn't open the Actions *panel in Flash at all! If this is the case and you don't want to mess with Exposé, pressing Option+F9 will open the* Actions *panel.*

However, those who were already hooked on the F9 shortcut before Apple introduced Exposé will have to change the keyboard shortcut for their little zoomy friend Exposé.

However you opened the Actions panel, this is how it should look when you open it for the first time:



Let's take a look at what we have here. At the bottom is a little tab that looks like this:



This tab tells you that any actions you add are going to be **attached to** a frame. You also know the actions will be attached to a layer called Layer 1, and on frame 1 of that layer via the text Layer 1 : 1 . . . um . . . you also know this must be true because that's the only frame you have at the moment!

Actions are always attached to something—usually a keyframe. (You can also attach a script to a movie clip or button using an older style of ActionScript coding, but you will *not* be using that in this book—it's outdated and not recommended anymore.)

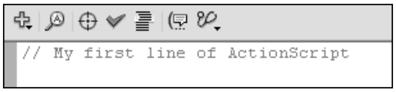
You tell Flash what you're attaching to by selecting it before you start typing in the Actions panel. The tab is very useful because it reminds you what you're attaching to and where it is. Sometimes this may not be obvious (because you have, for example, selected the wrong thing in error), and you should always get into the habit of checking the tab to make sure what Flash thinks it should be attaching to is what you want it to attach the script to. Most scripts will *not work properly* (or at all) if you get this wrong.

Sometimes you'll find that the script you're writing suddenly vanishes. This is one of those "Oh my God, I've lost all my stuff!" moments that every beginner has, but not to worry, it usually only means the keyframe you're attaching to has somehow become unselected, and something on the main stage is now selected instead. Before you start writing it all out again and glare at your cat/dog/co-worker for ruining your concentration and/or day, it usually helps to try reselecting the keyframe first.

Once you've selected a frame, you add scripts via the Script pane (the big white area to the right of the Actions panel). Simply click inside the Script pane and start typing. Type the following code and then press the Enter key:

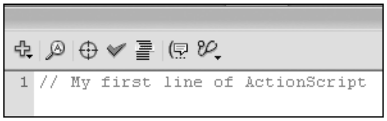
```
// My first line of ActionScript
```

Flash calls the little windowlike areas within a panel ***panes*** *(as in, "The panel is a window, and the bits inside it are the windowpanes").*



```
// My first line of ActionScript
```

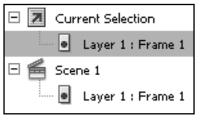
The // tells Flash that this is a **comment** rather than code, and Flash will actually ignore everything after the // until you press Enter. Adding comments to your scripts is a good way to remind yourself what the scripts do six months after you've written them and/or 20 minutes after you've written them, depending on how bad a day you're having. Press Enter on your keyboard to start a new line. Your scripts will consist of many such lines, and it's a good idea to *number* them so you know how far down a script you are (this also makes it easier for us to refer to line numbers later in the book). Select the little pop-up menu icon (at the top-right corner of the Actions panel) and select Line Numbers in the menu that appears.



```
1 // My first line of ActionScript
```

OK, let's look at the other goodies. On the left side are two more panes.

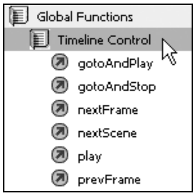
You'll notice that there's a little letter "a" above frame 1 on the timeline. This tells you that the keyframe here has a script attached to it. That's fine for small FLAs with not many timelines, but for bigger Flash productions, you need a centralized place that allows you to search out all your stuff that has ActionScript attached to it. The bottom-left pane, called the **Script navigator**, does just that. It consists of two file trees. The top one shows where the currently displayed script (in the Script pane) is within the Flash hierarchy, and the bottom one shows all scripts in the current FLA. Because you don't know much about the timeline hierarchy and how it affects ActionScript just yet, we'll leave further discussion of this pane until later, except to say that the reason the two trees look the same at the moment is because the one script you're looking at is the only one in the FLA.



```
⊟ Current Selection
    Layer 1 : Frame 1
⊟ Scene 1
    Layer 1 : Frame 1
```

The top-left panel (the **Actions toolbox**) is a list of little "book" icons, the first of which is called Global Functions. If you click it, the icon will "open" to reveal more books.

If you also open the Timeline Control book inside the Global Functions book, you'll see some icons:

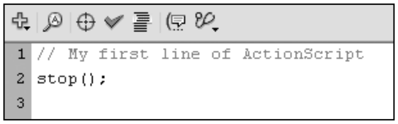


```
Global Functions
  Timeline Control
    gotoAndPlay
    gotoAndStop
    nextFrame
    nextScene
    play
    prevFrame
```

These icons represent **actions**, and the ones in the Timeline Control book are the most basic ones, so you'll be using them soon. The other books contain operators, functions, constants, and other bits and pieces you can use to tie lots of actions together in the way you want. If you hover the mouse pointer over one of them, a tooltip should appear that gives you a brief definition of the book's contents and where it's used.

Don't worry about the details of all these for now, though—you'll learn about them soon enough. This is the Actions toolbox, and it gives you a neat little way to write code in a no-brainer, "building blocks" fashion.

Look down the list of actions in the Timeline Control book and double-click the second-to-last one, stop. This adds a `stop()` action to your Script pane at line 2:



```
1 // My first line of ActionScript
2 stop();
3
```

You now have an action called `stop()` that is attached to the first frame on the layer called Layer 1. This action will stop the timeline at frame 1. Looking at the rest of the actions in the Timeline Control book, you'll have probably figured out that the `play()` action makes the timeline start playing, and so on.

Direct typing

Well, you've made a start. All the actions you're ever going to need are stashed away inside those icons, so you could continue using this technique to write all the code you're ever likely to need. What's more, working in this way will mean that there won't be any errors in your script (such as mistyping the `stop()` action as `slop()`), so you'll find it very hard to create errors in this way.

However, there's another option to consider: you can bypass the books completely and type your scripts directly into the Script pane.

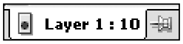
If you have a nervous disposition, this may seem like a worrying step. "It's only the first chapter, but already you're expecting me to type code and get it right first time?" you cry. But honestly, it's not like that at all! There are so many assistive elements when typing directly into the Actions panel that there's really no need to use the Actions toolbox. Although searching out the actions and other stuff in the toolbox seems like a safe option (you can't spell them incorrectly if you do it this way, and you have less to remember), it has one problem: it's darn slow! If you take the time to type everything in directly, except for the odd bit of code you can't remember, not only will you work faster, but also you'll get *much* faster with time.

OK, let's try it.

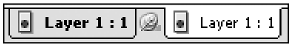
Don't worry about the parentheses and semicolon, `();`*, at the end of the* `stop` *for now. Sure, they make the simple* `stop` *appear a little otherworldly, and they look suspiciously like they were put there by someone with a serious code-fetish just to scare the nonbelievers off, but they're actually there for some simple reasons that you'll look at later on in the book.*

First of all, you want something that will tell you if you've made a mistake. Using the Script pane as you would a normal text editor, try changing `stop();` to `slop();` and you'll see that the first four letters have changed from blue to black. The Script pane will normally show all action names (or **keywords**) in blue, though you may want to use the ActionScript preferences (accessible from Actions ➤ Preferences) to change this color to a more noticeable shade of blue.

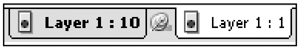
Second, you'll want to fix that "disappearing script" problem mentioned earlier. Just select frame 10 on the main timeline, add a keyframe (press F6), and select the keyframe to see what we mean—look, no more script! Of course, that's not quite true. Your script is still attached to frame 1, but you're looking at the script attached to frame 10, and there currently isn't one. You can see this if you remember to look at the tab at the bottom of the Script pane, though. It has changed:



So what if you want to work on the graphics on the stage at frame 10 while you're looking at the script in frame 1? Easy! Just click frame 1 again, and then click the pushpin button that's just to the right of the tab. This is called **pinning**. The tab won't disappear, even if you select something else. So now you're thinking, "Yeah, I see that, but now I have *two* tabs that show the same thing!"



The one on the right (the light one) is the one that's pinned, and the one on the left shows the frame or object currently selected. The confusing thing at the moment is that they're both showing the same thing. All becomes clear when you click back on frame 10. The tab on the left will change, and the pinned one to the right will stay as it is.



You can pin as many scripts as you need. Pinned scripts will go to the right, and the script for the currently selected item will appear to the left. Some applications with a similar tabbed system work the other way around, which may be confusing to some.

Pinning is a very useful feature to know about, but only so long as you remember to turn it off when you start trying to add actions elsewhere. Click the pushpin button again to unpin the script, and click back on the first frame. Now you're ready to continue.

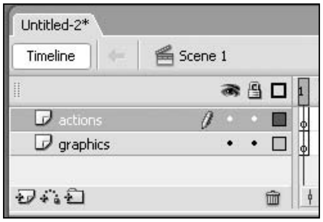
One more thing that's well worth knowing at the start about Flash is that it's **case sensitive**. When typing code, you generally have to get both the spelling *and* the capitalization right; otherwise, the code may not work.

The Actions panel will lie at the center of all the ActionScript that you'll ever write. You'll explore a few more of its mysteries as you work through the book, but you now have enough information to start and create your first ActionScript-enabled movie.

Several more helpful features are built into the Actions *panel—you've probably already noticed all the buttons running along the top of the* Script *pane. You won't examine them right away, though; it will be easier to demonstrate what they do once you have a little more scripting experience under your belt.*

Let's start with a simple example that shows how easily you can bypass the usual flow of a movie with just a tiny bit of ActionScript. You can close the current FLA if you wish (you don't have to save it).

1. Create a new Flash Document movie and add a new layer to the root timeline. Rename the original layer as graphics and add a new layer called actions. This is a good way to start all your ActionScripted movies.



Rather than use the File ➤ New *menu option, you can right-click/Cmd-click the tabs above the timeline to open/close/save files—this is much quicker!*

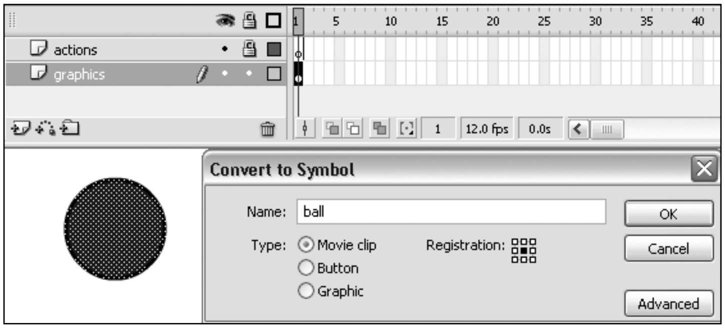
Scripts should normally always be attached to keyframes on the topmost layer (if you put them on the bottommost layer, the scripts may not work on the Web because then they'll be loaded before the graphics, and this tends to confuse Flash). It's also a good idea to have only scripts in the topmost layer, which is why you'll call this layer actions—to remind you that that's what this layer contains, and nothing else. As you progress through the book, you'll quickly become familiar with the idea of having an actions layer residing at the top of the timeline. Naming a script layer actions is a very common convention among Flashers.

> *A good trick while you're working is to always keep the* actions *layer locked. This stops you from inadvertently placing graphics in this layer, but it doesn't stop you from attaching scripts to it or modifying the frames and keyframes in the* Timeline *panel. Locking a layer simply prevents you from editing the contents of the stage.*

2. Lock the actions layer (for the reason just noted).

Let's add those graphics now. You don't need any terribly fancy graphics to demonstrate what's going on with the timeline. For simplicity's sake, let's just create a simple motion tween between two points.

3. Select frame 1 on the graphics layer, and use the Oval tool to draw a circle on the left side of the stage. With the Selection tool, select the oval (double-click it on the fill to select both the fill and the outline stroke) and press F8 to make it into a symbol. The Convert to Symbol dialog box will appear. Select the Movie clip option and call it ball.

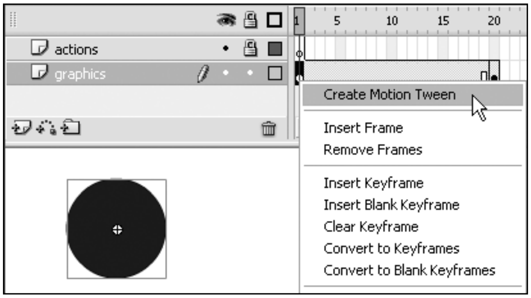


Convert to Symbol

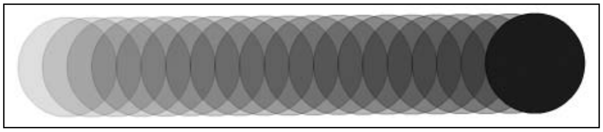
Name: ball

Type: ● Movie clip ○ Button ○ Graphic

4. Now select frame 20 of the graphics layer and press F6 to insert a keyframe. Right-click (or Ctrl-click on the Mac) frame 1, and select Create Motion Tween from the menu that appears.



5. To complete the animation, select the keyframe at frame 20, and then move `ball` to the right to create a motion tween. If you test your creation with Control ➤ Test Movie, you'll see the ubiquitous "ball moves forever from left to right" animation. So much for originality!

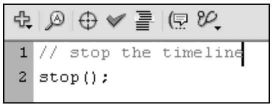


The ball's movement depends on the timeline, and you know that ActionScript can be used to control that. Let's prove it.

6. Select frame 1 of the actions layer and take a look at the Actions panel. Make sure the Script pane is active (click in it to give it focus), and you should see the cursor blinking away right next to it. Also, note that the tab at the bottom of the Script pane reads actions : 1. You're about to begin your hand-coding journey, so flex your fingers, crack those knuckles, and type exactly what you saw before:

```
stop();
```

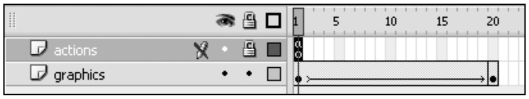
7. Place the cursor at the beginning of line 1, press Enter to shift the action down to line 2, and fill the now-blank first line as shown here to remind you what your code does with a comment:



```
1 // stop the timeline
2 stop();
```

As mentioned earlier, when the movie runs, Flash will completely ignore all the gray text (everything following the double slash), so you can add as many comments as you like. In fact, we strongly recommend that you do just that!

8. If you take a look at the timeline, you'll see that there's now a little "a" showing in the first frame. This reminds you that there are actions attached to this frame.



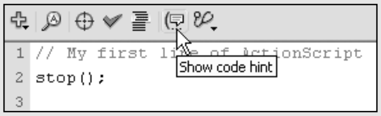
9. Run the movie again and you'll see what a difference your single action has already made. As expected, the circle doesn't move. That's all well and good, but on its own, this isn't really that exciting, let alone useful. You're some way into this chapter, and all you've done with ActionScript is stop your movie from doing anything at all. OK, it's time we introduced a couple more actions before *you* decide it's time to stop.

10. Head back to the Actions panel and delete all the script you've written so far. Replace it with the following lines:

```
// jump to frame 20 and then stop
gotoAndStop(20);
```

11. Run the movie again, and you'll see once again that the ball is dead still. This time, though, it's dead still on the right side of the stage, because you've told the timeline to go to frame 20 and then stop.

> *You may have noticed that just after you entered the opening parenthesis, a tooltip popped up showing you the full command, but it swiftly disappeared again when you entered the closing parenthesis. This is another of the* Actions *panel's helping hands, called a* ***code hint****. This is Flash's way of telling you that it's recognized what you're typing, and that all you need to enter now is a closing parenthesis and a semicolon to finish off the job.*
>
> *It may not look like much now, but when you start looking at more complicated actions later on, you'll begin to find it invaluable. If you ever want to see the code hint again, you can click between the parenthesis at the end of the action and click the* Show Code Hint *button at the top of the panel.*



```
// My first line of ActionScript
stop();
```

12. Now select frame 20 on the actions layer, press F6 to insert a keyframe, and attach these lines of script. You don't have to unlock the layer to do any of this, the advantage of this being that if you accidentally on purpose click the stage during this part of the tutorial, you'll see that your keyframe doesn't become unselected (if you do try to draw or affect a locked layer, you'll see a pop-up asking whether or not you want to unlock the layer; click No). The beauty of this is that you won't inadvertently get to the "Dude, where's my script?" situation.

```
// jump to frame 10 and then play
gotoAndPlay(10);
```

Now you should have a little more action to see on the stage. When the movie starts up, it will see the `gotoAndStop()` action you've attached to frame 1. This tells it to jump ahead to frame 20 and stop, so it will do just that.

However, attached to frame 20 is an action telling your movie to jump back to frame 10 and start playing again. Even though the movie will have stopped playing at this stage (i.e., it won't be moving automatically from one frame to the next), it still has to follow instructions.

> *You can see the movement of the movie in terms of frames if you have the Bandwidth Profiler open during the test (*View ➤ Bandwidth Profiler*).*

So, you see the circle jump back to the middle of the stage and start to move smoothly across to the right. Of course, once you reach frame 20 again, you're sent back to play from frame 10, and so on, and so on, for as long as you care to watch it.

Hmm. That's a little more interesting to look at, but it still doesn't seem like much to write home about. Why not just start the tween from the middle of the stage? Surely the end result is the same. Maybe, but the important thing you've seen here is that you can use a couple of simple actions to *override what would normally happen* in the movie. In this case, you've stopped the root timeline and jumped about from frame to frame.

This is just the tip of the iceberg. There's plenty more in a movie that you can turn to your bidding.

Who are you talking to?

One of the most important things to understand about actions is that they always have a **target**. If you think of them as commands again, it's fairly obvious that each command is directed at someone or something. For example, when Kristian's parents would tell him to go clean his bedroom, they might just say, "Get up there and clean that darned bedroom *now*!"

Now that would be fine if Kristian was the only person in the room, but if other people were there too, they'd need to say, "*Kristian*, get up there and clean that darned bedroom *now*!"

So what's all this got to do with ActionScript? Well, you already know that actions are basically just commands to the Flash movie, telling it to do something. You know that they're always attached to something, and so far that's always been a keyframe on the root timeline. When you attached a simple `stop()` action to the root timeline, it stopped *the current timeline*, where *current* is the timeline the code is on. That being the case, how do you target an action at something else? What, for that matter, do you do when you want a script to control *two or more* movie clips?

Rather like when we give each other instructions, Flash will sometimes *assume* what our scripts are supposed to control, and sometimes it will need us to tell it via a name.

> *The movie clip that a line of script controls by default is called the **scope** of that script. Although this is a slight simplification of what scope actually is, it's a good approximation to use at this stage.*

When you used the `stop()` action earlier, you used the **assumed** scope—the timeline the code is attached to. As you get into more complicated situations, the process becomes much like human interaction. When you have a *choice* of who to talk to (or you simply want to make it clear), you have to use *names*. ActionScript calls these names **instance names**.

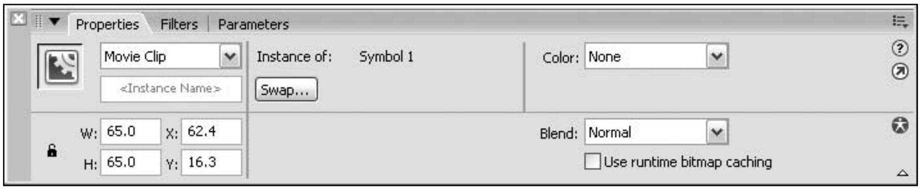
Controlling movie clips on the stage

It's all very well *saying* that, but how do you *do* it? Well, the answer comes in two parts.

First, you need to give the movie clip a name. "But it already has a name!" we hear you cry. When you create a movie clip (or any kind of symbol), it appears in the Library panel with a name associated with it, normally "Symbol 1" or something like that, unless you change it to something else. In fact, this name doesn't mean anything at all to ActionScript; it's just a label that Flash gives the movie clip to make life easier when you're making movies. What's more, the items you see in the Library aren't actually part of the movie—it's only when you drag an **instance** of the symbol onto the stage that it becomes a thing you can control with ActionScript.

> *One reason you can't use the Library name as the instance name is because if you dragged two instances of the same symbol onto the stage, they would have the same name!*
>
> *Some of you might be wondering at this stage, "How did I control the previous ball animation without giving the ball instance a name?" Well, the simple answer is that you were never controlling the ball. You were controlling the tween on the main timeline (that includes the ball). Because the main timeline was the implied scope of your animation, you could get away with not stating an instance name.*

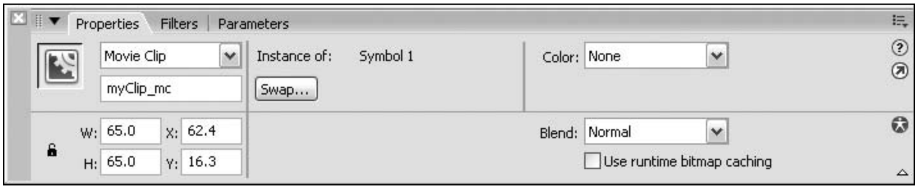
Once you've dragged your symbol onto the stage, the Property inspector gives you a neat little way to designate that instance with its very own unique name. Say you have a movie clip in the Library called Symbol 1, and you've just dragged it onto the stage. This is what the Property inspector might look like:



Movie Clip
<Instance Name>
Instance of: Symbol 1
Color: None
W: 65.0 X: 62.4
H: 65.0 Y: 16.3
Blend: Normal
□ Use runtime bitmap caching

There's a text box just below where Movie Clip is selected in the drop-down box that holds the following text in gray: <Instance Name>.

No prizes for guessing, but this is where you give your movie clip instance a name. Let's assume for now that you call it `myClip_mc`:



Movie Clip
myClip_mc
Instance of: Symbol 1
Color: None
W: 65.0 X: 62.4
H: 65.0 Y: 16.3
Blend: Normal
□ Use runtime bitmap caching

There are a couple of things that you may be worrying about here:

- You may be wondering why you call the movie clip "myClip" instead of "my clip". Computer languages don't like spaces in any names so, although you're allowed to add spaces in the Library name (because it will be read by you only), you can't have spaces in the instance name (because ActionScript uses it). One way around the fact that you can't use spaces is to use **camelCase**, where you put a capital letter after the point at which you would normally use a space (it's called camelCase because of the hump the change of case causes).

- You may be wondering what the `_mc` suffix is all about. Why not just call it `myClip`? Humans have names for boys and girls, so that you can (usually) tell the sex of someone without actually seeing the person. Unfortunately, there are no such names for movie clips, buttons, and all the other things in Flash that can have a name, so the Flash parser (the code that interprets your code) can't tell if the name you give the movie clip is a typical name for a movie clip. The way around this is to use a suffix. The `_mc` tells Flash "And it's a new, bouncing baby movie clip!" in the same way most of us know that baby John would be a baby boy.

Don't worry if camelCase looks like something it will take a while to get used to—it will be second nature by the time you finish the book, honest!

You'll also soon come to appreciate how useful it is to have the `_mc` reminder like this, so that when you see it in code, you know immediately that it's referring to a movie clip. What's more, the Actions panel also recognizes `_mc` instance names as movie clips, so it can prompt you with options that are specifically relevant to movie clips. You'll see the use of suffixes come into play in more detail later in the book, but here's a quick list of all of the suffixes that Flash recognizes and what they mean:

Suffix	Description
`_mc`	Movie clip
`_btn`	Button
`_sound`	Sound
`_txt`	Text field
`_str`	String
`_array`	Array
`_video`	Video
`_date`	Date
`_color`	Color
`_lv`	Load vars
`_xml`	XML

For a full list of code suffixes, search for "About using suffixes to trigger code hints" in Flash's Help *panel.*

The first two suffixes are the ones you'll use most often, but it's worth knowing what the others are so that you'll recognize them when you see them.

Now that your instance has a name, you can use ActionScript to talk to it. This is where the second part comes in: **dot notation**. The **dot** (aka the **period** or **full stop**) might be the smallest text character going, but when it comes to ActionScript, it's also one of the most important. This is how you use it to tell Flash that you want to stop the timeline of the movie clip instance called `myClip_mc`:

```
myClip_mc.stop();
```

In terms of the earlier room-tidying analogy, "Kristian, go tidy your room!" would be the following in Flash, if there was some ActionScript to force him to do it (luckily, his parents never asked for it via the Flash wish-list e-mail, so we don't think there is):

```
kristian.goTidyRoom(your);
```

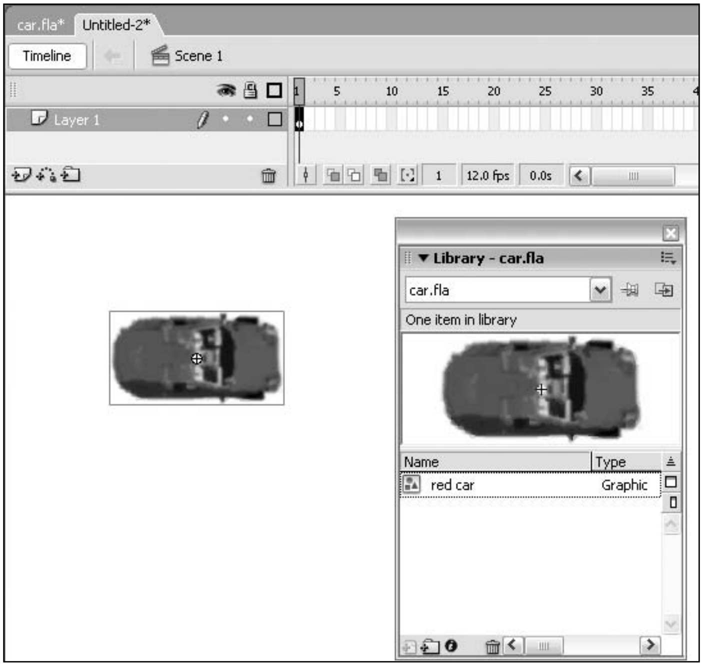
The bit before the dot tells Flash who (rather, what*) the instruction is directed toward. The bit after the dot is what you want to be done.*

Let's bolt together all that you've learned so far.

Talking to different timelines

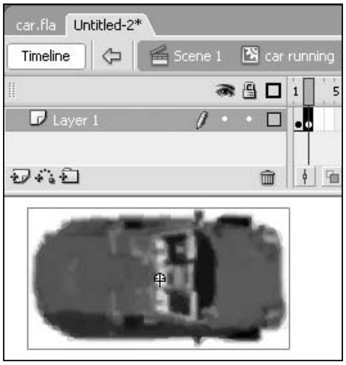
In this exercise, you're going to use ActionScript to control a couple of cars. You can find the graphic symbols we've used in the download for this chapter (in the file `car.fla`), though there's nothing stopping you from creating your own!

1. Open `car.fla` and then create a new Flash document (File ➤ New). With the new Flash document selected, open the Library panel. In the Library panel, select car.fla from the drop-down list and drag an instance of red car to the right side of the stage:

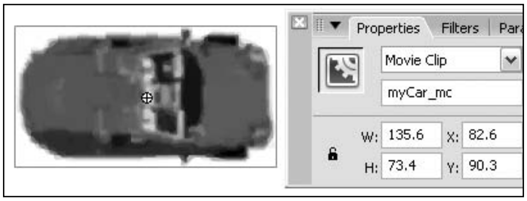


2. Press F8 to convert the car to a movie clip symbol called `car running`.

3. Double-click the unnamed instance (the car) to edit the movie clip in place. Select frame 2 of the timeline and press F6 to insert a keyframe. Select the graphic again and use the Property inspector to move it vertically (i.e., change its y-coordinate) by 1 pixel. If you run the movie now, you'll see the car shudder slightly up and down, which tells you that the car's engine is running.



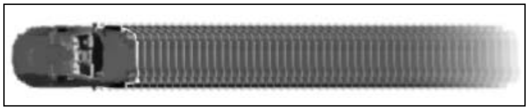
4. Click Scene 1 above the timeline to return to the main timeline. Make sure the instance of `car running` is selected and use the Property inspector to set its instance name to `myCar_mc`.



Movie Clip
myCar_mc
W: 135.6 X: 82.6
H: 73.4 Y: 90.3

5. Select frame 50 of the main timeline and press F6 to insert a keyframe. Select frame 1, followed by Insert ➤ Timeline ➤ Create Motion Tween to set a motion tween between frames 1 and 50.

6. Next, hold down Shift while you drag the car across to the left side of the stage. Run the movie again, and you'll see the car shudder its way across the stage. So far, so good!



Now it's time to add some ActionScript, so create a new layer called actions, lock it, and then select frame 1 in this new layer.

7. Open the Actions panel and enter the following script:

```
// stop the car traveling
stop();
```

Since you've attached your script to the root timeline, Flash assumes that the `stop()` action is directed at the root timeline. That's where your car's motion is controlled, so the car should stop traveling.

Test the movie and you'll see this is true. More importantly, though, while the car may not actually be going anywhere, its engine still appears to be running. That's because the shuddering motion is part of the movie clip, so it's controlled by a separate timeline.

8. One way you could stop the car's timeline would be to open up the movie clip and attach a `stop()` action to its timeline. That would do the trick, but it would quickly get tricky if you wanted to do more complex things and found yourself having to switch back and forth between the timelines. There's no point, anyway, since you have a perfectly good way to do the same thing directly from the root timeline. Change your script so that it looks like this:

   ```
   // stop the car's engine
   myCar_mc.stop();
   ```

   and try the movie again. You'll now see that the car is moving across the screen, but it's no longer shaking. Someone has either fixed the shaky engine or left it out of gear and forgotten to put on the brake!

In terms of your movie, you've now stopped the `myCar_mc` timeline but left the root timeline alone. If you stopped both of them, the car wouldn't move at all:

```
// stop the car moving at all
stop();
myCar_mc.stop();
```

The other big advantage of using dot notation and scripting everything from the main timeline is that you can now do different things to different instances. The process of identifying different movie clips for different actions is called **scoping**. The first `stop()` is scoping the main timeline, and that's what it stops. The second one is scoping `myCar_mc`, and unsurprisingly, it stops the shuddering animation caused by that timeline.

9. Create a new layer and add a new instance of `car running`, giving it the instance name `myCar2_mc`. You may want to rename these layers as car 1 and car 2. Make sure your new car is well below the original and re-create the motion tween described earlier on. Run the movie, and you'll see that the first car is still dead, but the second car is running (even if it's not going anywhere right now).
10. Finally, let's prove that it's not just left to chance that one car is running while the other isn't. Change the last line of your script to read as follows:

    ```
    stop();
    myCar2_mc.stop();
    ```

11. Run the movie one last time, and sure enough, the engine in the first car (`myCar_mc`) now runs, while the second (`myCar2_mc`) is motionless.

Arguments

Of course, Kristian's parents had an easy time of it with the bedroom-cleaning instructions. He had only one bedroom, so it was obvious which one he had to clean. If he'd been one of the Getty children, his parents might have had to say, "John-Paul, get up to bedroom number seven this instant or it's no rum truffles for you!"

Kristian's parents didn't need to add extra details about *which* room they meant, because it was obviously *his* room. John-Paul's parents had to add an extra bit of information: *which one* of his bedrooms he had to clean.

This extra piece of information that qualifies the initial command is called an **argument**. No, this doesn't mean that John-Paul refused to clean his room and started a family fight. It's just a name we give to values that help to determine what an action does for us.

So, to tell Kristian to tidy his room via a line of ActionScript, you would have (as you saw earlier) this:

```
kristian.goTidyRoom(your);
```

Most people have only one room, so in most cases, the room you have to clean is implicit—yours. This implicit argument is often assumed in many actions if you give no argument as the *default* argument. For example, the following:

```
kristian.goTidyRoom();
```

would mean the same thing as the previous line of ActionScript.

For young John-Paul, the situation is slightly different. If you wanted him to clean bedroom number 7, you might use

```
johnPaul.goTidyRoom(7);
```

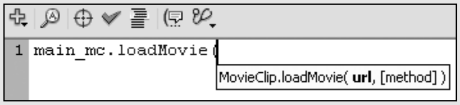
The argument here becomes important because John-Paul has no default bedroom.

Finally, if Kristian wasn't bad enough to have to tidy his room, but was due for a temporary grounding, he might be told to simply go upstairs:

```
kristian.goUpstairs();
```

Here, no argument is needed because what Kristian has to do is simple: go upstairs and make himself scarce. Notice that even here the parentheses, ( ), are used, even though nothing is expected inside them. This is because the `instancename.action(argument);` code structure is *standard*; you must always use it even if some of it isn't required, rather like a period isn't always needed for total comprehension (like at the end of this sentence), but you put one in anyway.

When Flash requires arguments for a command from you, it will very conveniently tell you exactly what it wants. This information will appear as a code hint. So, in the event that Flash recognizes a command you have typed in, a code hint will appear displaying the command and the arguments you need to supply (if any). Even though this example is a little advanced for this chapter, here's the information required for `loadMovie`*:*



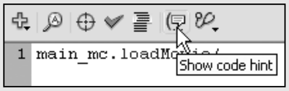
```
main_mc.loadMovie(
```

MovieClip.loadMovie(**url**, [method])

The `loadMovie` *command requires two arguments as shown in the code hint tooltip. The first,* `url`*, is a required argument that will specify the file for Flash to load in. The second,* `method`*, is displayed in square brackets because it is an optional argument. The* `url` *argument is highlighted in bold because it is the next piece of information Flash requires from you.*

As you progress through this book, you'll find code hints invaluable in helping you decipher what information you need to provide Flash with, so that it can do its job properly. Code hints are not meant to be temporary stabilizers; rather, they are intended to provide a permanent reference to an ever-growing library of Flash commands (and if you're the forgetful type, you'll become very friendly with the little fellas!).

One last tip before we move on: If you ever require a code hint when it isn't shown, click the Show Code Hint *button in the* Actions *panel, and the code hint will reappear.*



```
main_mc.loadMovie(
```

Further, sometimes Kristian will be the only misbehaving kid in the room, so the "kristian" part isn't needed—there's no confusion in omitting the name, because everyone knows you mean "the ornery child in this room":

```
goUpstairs();
```

This all goes some way toward explaining

- Why you always have `stop()` rather than simply `stop`. You *must* have the parentheses, `( )`, because they're part of the basic structure of the language, irrespective of whether any arguments are ever actually needed.
- Why you sometimes have `stop()` (meaning the timeline this action is attached to, and there's only one of those, so you don't need to say anything else) and you sometimes have `myClip_mc.stop()` (because you don't want to stop the default timeline, but another one, which you have to name).
- Why some actions *always* have an argument, such as `gotoAndPlay()` (an argument is always needed because there's no implied frame to `goto`; you always have to say, for example, `gotoAndPlay(7);`).

Listening to what your movies are telling you

So far, all the ActionScript you've looked at has been fairly predictable and not terribly exciting. That's because you've figured everything out in advance—before you even run the movie, you know exactly what actions you want to give your movie and exactly when you want them to kick in.

When you think of Flash, you probably think of **interactivity**. Whoever ends up watching your movie should be able to react to what they see and hear, and make things happen in the movie spontaneously. That's probably why you're here in the first place, learning how to add some more of that interactive power to your designs.

Interactivity is a two-way process, though. As well as giving out commands, you need some way to respond to what's going on when the movie runs. You need some way to **trigger** actions whenever some event happens while the movie is running—for example, when the movie finishes loading, when the user clicks a button, when the hero of your adventure game catches all the magic artichokes before the villain's bomb goes off . . . you probably get the idea.

As an example, when Sham was a child, he was given a cuckoo clock as a present. Well, actually, a cuckoo clock only in the loosest sense, given that it was in the shape of a cartoon lion with a swinging tail as a pendulum. Every time the clock struck the hour, the lion would wake up, open his mouth, and growl. He could act as an alarm clock, too, and give an almighty roar when the alarm went off.

There were two things that set the lion into action:

- When the clock reached the beginning of the hour, it was time to growl.
- When the alarm time arrived, it was time to roar.

So, in this case there were two **events** that the lion was waiting for. As soon as either occurred, the lion would perform the appropriate **action** (a growl or a roar).

In fact, the magic word here is **event**. You need a way to write ActionScript that can react to events that happen during the course of the movie. That is, "Whenever *this event* happens, perform *these actions*." Or, to be more succinct, "If *this* happens, do *that*."

> *ActionScript is called an **event-driven language**, which sounds like complicated programming-speak until you realize that, just like the lion, it's set up to wait for something to happen and then react to it—nothing more complicated than that.*

Events in Flash

So what sort of event happens in a Flash movie? Well, it could be something obvious that happens externally—say, the user clicking a button on your website or typing on the keyboard. An event can also be something less obvious that happens internally within Flash, like loading a movie clip or moving to the next frame on the timeline.

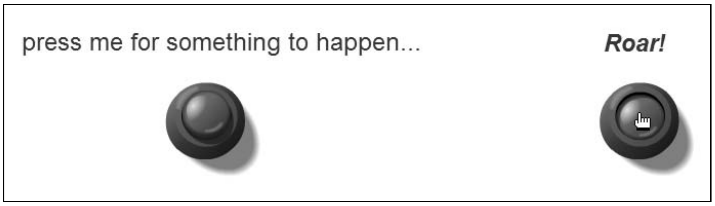
Whether you're dealing with an internal event or an external event, you can write a sequence of actions (called an **event handler**) that will run whenever that event occurs.

In the real world, you can see all sorts of event/event handler pairs around you. For example, your electric kettle waits for the water to start boiling (the event) and has a special circuit that switches the power off (the event handler) when this occurs. Another example would be your central heating. When the air in your apartment falls to a certain temperature (the event, in which this temperature is decided by how you've set the thermostat), your heating kicks in (the event handler) to make things warm and toasty again.

External events

Back in the Flash world, let's consider the simplest event/event handler partnership: a button click.

1. The button push_btn sits there, waiting for some user interaction.
2. The user clicks push_btn, and this shows up as a Flash event called onPress.
3. Flash looks for an event handler that's been assigned to that particular event on that particular button: push_btn.onPress.
4. If Flash finds one, it runs the ActionScript inside it; otherwise, it does nothing.



The most important thing to notice about events is that they're very specific to the thing that generated them. Say you have another button called red_btn. If you click it, it also generates an onPress event, but it doesn't trigger anything unless you specifically define an event handler for red_btn.onPress.

Yes, the dot notation gets everywhere! Don't panic, though, the rules are similar to what we covered before: the bit before the dot tells Flash where the bit after the dot is *coming from*. It's really just as it was with actions, except that it's telling you *where an event came from* rather than *where an action is going*.

Internal events

This event/event handler relationship is fairly easy to spot when we're talking about external events such as a button click. However, internal events aren't always so obvious, since movies generate them automatically. Probably the easiest one to get your head around is the onEnterFrame event, which is generated every time a timeline's playhead enters a new frame. Yeah, but what does this mean in practical terms? Well, it means that it's easy to write a bit of script that Flash will then run for *every* frame.

Say you want to make the timeline run at double speed. _currentframe refers to the number of the frame you're currently on, so you could write an onEnterFrame event handler that contains the action gotoAndStop(_currentframe+2):

1. The movie starts, generating an onEnterFrame event.
2. This triggers gotoAndStop(1+2), sending you to frame 3.

3. This generates another `onEnterFrame` event.
4. This triggers `gotoAndStop(3+2)`, sending you on to frame 5.
5. And so on . . .

Basically, you skip every other frame, so the movie appears to run at twice the normal speed. If you did the same thing using `gotoAndStop(_currentframe-1)`, you could even make the movie run backward, as every time the playhead enters a frame, it would be told to go to the frame before the one it's currently in, where it would find the same command again.

OK, this may not seem like particularly useful stuff just at the moment, but that's just because you're still fooling around with the timeline. Once you've tackled a few more actions, you'll start to realize just how powerful this `onEnterFrame` business really is.

Before doing that, though, let's get down to business and look at how you actually write these fabled event handlers.

Introducing event handlers

Earlier on, we described event-driven programming in these terms: "Whenever *this event* happens, perform *these actions*." You now know that *this event* might be something like `push_btn.onPress` and that *these actions* are what you call the event handler. You can express that in ActionScript like this:

```
push_btn.onPress = function() {
  // actions here make up the event handler
};
```

Hmm. It's not nearly as intuitive as `stop()` and `play()` were—we confess, there's a bit more to this one than we've talked about so far. But before you shrug your shoulders and start checking the index for what `=` and `function` are all about (because we know you want to know!), just stop for a moment. You'll move on to all that soon enough. All that matters here is this:

> *Whatever actions you write between the curly braces* { *and* } *will be run whenever someone clicks the button called* `push_btn`.

While it may look a bit intimidating at first, you'll forget all about that once you've used it a couple of times—event handlers will almost always look much the same.

Nesting spiders—argh!

Just like some people hate spiders because they have too many legs, some programmers hate lines with too many brackets and commas and stuff. These are easier to come to grips with in ActionScript once you realize that these aren't the sorts of brackets that require you to reach for a calculator and start thinking; they're just there to help arrange things into groups, like this:

- All small dogs (including dachshunds and terriers) should go to the tree on the left, but all other dogs (including beagles and Alsatians) should go to the tree on the right.

This looks a *lot* less frightening than

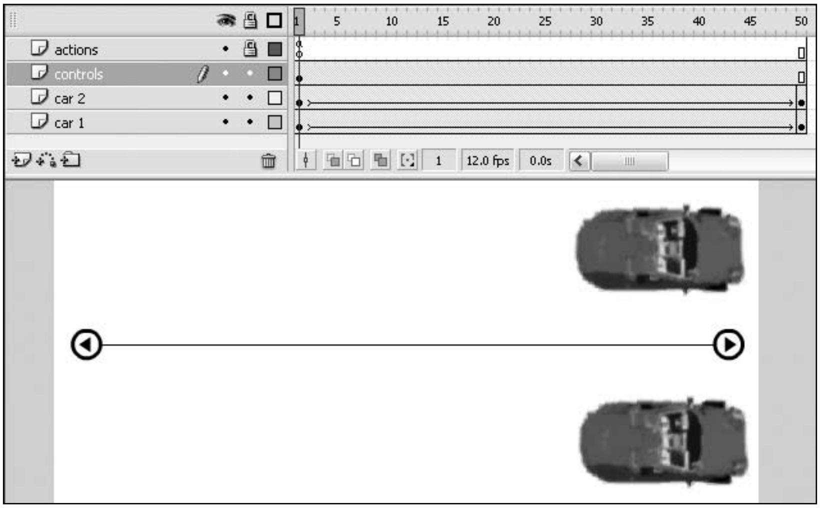
$$s = {}^{1}/_{2}(u+v)t + 5(a+20c) - 1$$

Ugh! Spiders! We know a lot of the stuff coming up will look like that awful equation, but it's not as bad as all that. Most of the time, it's just a shorthand way of writing the "dogs and trees" sentence. When any sort of math gets involved, it's usually of the 1+ 1 = 2 variety.

Let's get over any apprehension about event handlers as quickly as possible, and put them through their paces. You're going to base this example on the earlier FLA you made with the racing cars going across the screen, so either open your own copy or grab the file called `the_race.fla` from this chapter's download folder.

You're going to begin by adding some buttons to the stage, so you may want to close the Actions panel for the time being.

1. Create a new layer called controls, and place it above the car layers.
2. Now, rather than creating buttons from scratch, pull some good-looking ones from the Flash Buttons Library (Window ➤ Common Libraries ➤ Buttons). Open the Classic Buttons folder, followed by the Circle Buttons folder, and drag one circle button - next and one circle button - previous button onto the stage.
3. Drop the buttons at either side of the stage, and add a horizontal line between them:



4. Select the right button and pop down to the Property inspector to give it the instance name back_btn. Do the same for the left button, calling it forward_btn. Any guesses for what they're going to do?

5. OK, it's time to open the Actions panel again. Make sure that frame 1 of the actions layer is selected, delete the existing script, and enter the following:

```
// stop the cars traveling
stop();
// event handler for onPress event on 'back' button
back_btn.onPress = function() {
  // goto the previous frame and stop
  gotoAndStop(_currentframe-1);
};
// event handler for onPress event on 'forward' button
forward_btn.onPress = function() {
  // goto the next frame and stop
  gotoAndStop(_currentframe+1);
};
```

Did you spot the code hint when you typed in the gotoAndStop *commands?*

The code looks a bit intimidating, but actually, there's less script there than it might seem, mainly because of all the comments we've put in. Let's consider it one piece at a time:

- You begin by telling the root timeline to stop in its tracks. Otherwise, the cars would zoom straight across the stage and there would be nothing for you to do.
- Next, there's an event handler, which listens out for an onPress event coming from the "back" button. If it picks one up (as it will if someone clicks the button), then you use the gotoAndStop() action to jump to the previous frame.
- The next event handler is pretty much the same, except it listens out for an onPress event coming from the "forward" button and jumps to the next frame on picking one up.
- Each time you write an event and event handler, you'll notice that after you type the first { brace, Flash will start to indent your script. Likewise, just as soon as you type the closing brace }, it will automatically "outdent" back to where it started. This incredibly useful feature is called **auto-formatting**, and it can be a real lifesaver, especially later on when you start putting braces inside braces.

If the formatting of your code ever goes a bit awry for some reason, you can easily bring it back into line by clicking the Auto format *button in the* Actions *panel.*

6. Now run the movie, and you should see a couple of motionless cars. Been there, done that. So what? Try clicking the "forward" button, and you'll see both cars inch forward. Then click the "back" button and both cars will move back again.

Yes, with just a couple of really-not-so-scary event handlers, you've wired up a couple of buttons that give you complete control over the movie's main timeline. Not bad, eh?

Now you may feel the need to point out that with both cars covering exactly the same ground, it's really not much of a race. Yes, the foolish and shortsighted approach we used for animating the cars (two motion tweens on the same timeline—d'oh!) doesn't really do the movie any favors.

What if there was a quick and simple way to separate them? (Preferably involving ActionScript and not too many long words . . .)

Animating movie clips with ActionScript

OK, this isn't strictly part of the chapter's storyline, but we're not going to leave you wondering for the sake of a lesson plan. After all, this is supposed to be fun!

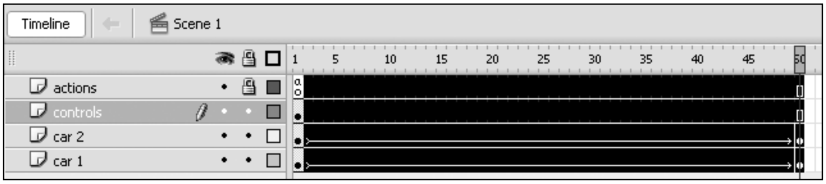
So far, you've done all your animation using tweens, and all your actions have been aimed at one timeline or another: stopping, starting, jumping about the place, and so on. That's a pretty good place to start if you're already quite familiar with Flash, since you're mainly tackling things you're already aware of.

Now that you've begun to grasp ActionScript stuff like targeting actions at movie clips and wiring up event handlers, we can start to reveal the truth . . .

You don't need the timeline to animate a movie clip.

Rather than explaining that statement, we're going to show you the alternative.

1. Back at the races, click frame 2 in the actions layer and drag down to frame 50 in the car 1 layer, so that frames 2 to 50 are selected on every one of the four layers:



2. Now right-click (or Ctrl-click) the selection to call up the context menu, and select Remove Frames to get rid of everything following frame 1. Yes, that's right, you want to lose *everything* except the contents of frame 1 in each layer.
3. Select frame 1 in the actions layer and open the Actions panel. You'll need to make the following highlighted changes to get the first set of buttons working:

```
// stop the cars traveling
stop();
// event handler for onPress event on 'back' button
back_btn.onPress = function() {
  //move myCar_mc horizontally +10 pixels
  myCar_mc._x += 10;
};
```

```
// event handler for onPress event on 'forward' button
forward_btn.onPress = function() {
  //move myCar_mc horizontally -10 pixels
  myCar_mc._x -= 10;
};
```

Test the movie now, and you'll find that the top set of buttons controls the top car, while the bottom car sits still. Let's fix that.

4. Now add the following script to the end of what's already there:

```
// event handler for onEnterFrame event
onEnterFrame = function() {
  // move the myCar2_mc across to mouse position
  myCar2_mc._x = _xmouse;
};
```

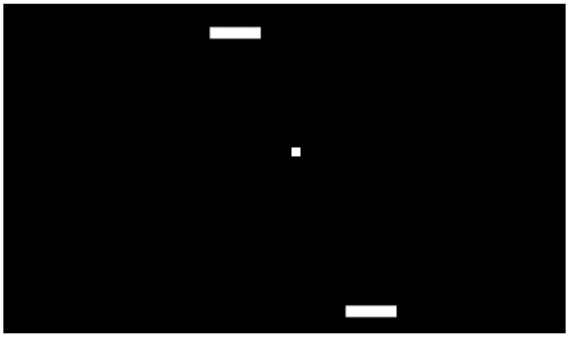
This lets you control the second car using the mouse. Just as `_currentframe` refers to the current frame number, `_xmouse` refers to the mouse's current x-position, and you use this to update the car's position. Because it's inside the `onEnterFrame` callback, it's updated 12 times every second (this is the frame rate of the movie), which keeps the position up to date.

5. Run the movie again. Now you can use the two sets of controls to maneuver each car separately.

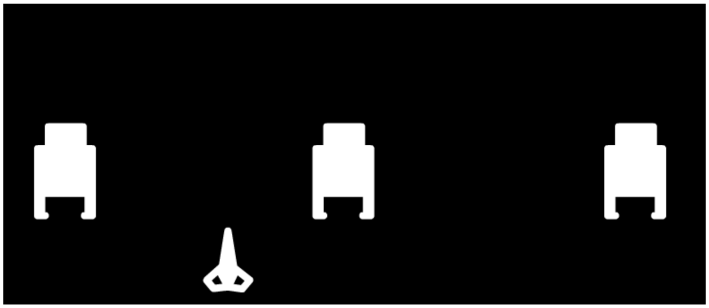
This animated effect is far more advanced than anything you've done thus far in this chapter—it's truly interactive. The original car race movie just moved the cars from left to right, and that's all it will ever do. This time, you can control the cars using buttons and the mouse.

Even simple interactivity like this opens up fantastic possibilities that you didn't have before. With a little imagination, you could replace each of the cars with a bat, and bounce little squares back and forth across the screen, just like one of those old TV video-game consoles.

Tennis, anyone? Take a look at `pong.fla` in this chapter's download to see how easy it is to create an interactive bat and the beginnings of a game.



As another option (perhaps just a *little* more exciting), you could change one of the cars into a spaceship, so it could be The Lone Crusader on a mission to save the world from the evil alien hordes.



This time, your Starfleet SpaceCruiser has to dodge alien plasma bolts. *Save those cities, young pilot!* Have a look at `spaceinvaders.fla` from the download if you want to see this screen moving.

You're not yet *quite* advanced enough to start adding alien hordes or the associated "bloolly bloolly bloolly blooop" sound effects just now. But hey, you've only been learning this stuff for about 20 minutes!

Remember that all this movement comes from just a few lines of ActionScript. Even better, they're the same lines of ActionScript every time, and the whole thing is only one frame long. You're already up to the maximum interactivity offered by the home computer entertainment systems available in the early 1980s—just think what you could achieve with another few frames and 40 more lines of ActionScript.

This is why it pays to know ActionScript. It's so much more versatile and dynamic than plain Flash that it quickly leaves traditional Flash animation standing in the corner feeling sorry for itself!

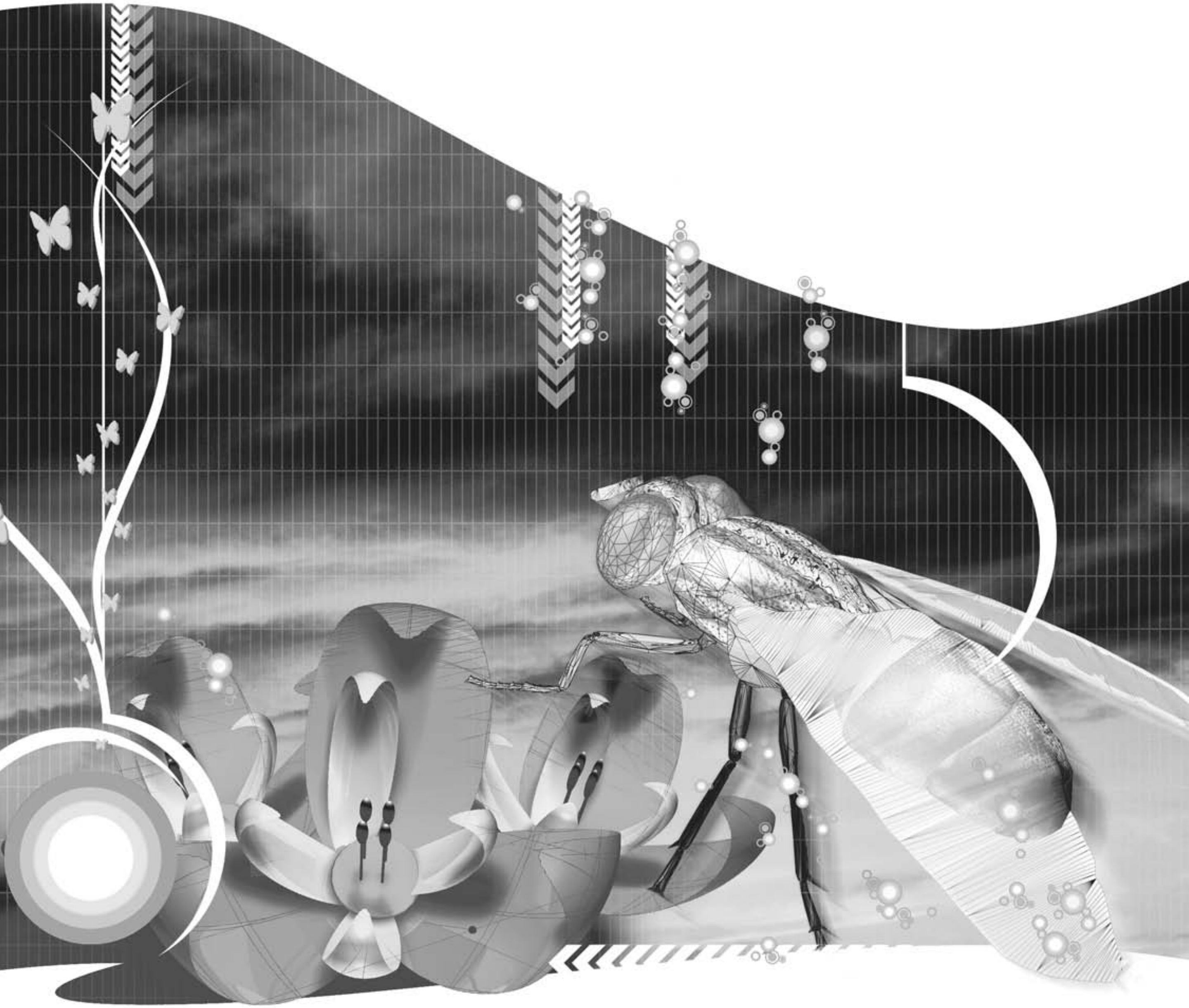
Summary

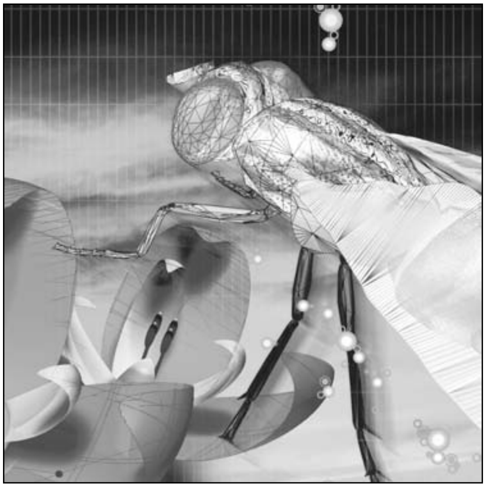
In this chapter, you've gone quickly through the basic concepts behind event-driven programming. You started off by looking at how to attach script to the root timeline, and then you saw how to write **actions** that control the timeline. Along the way, you learned about **targets** and **instance names**.

After that, we introduced the idea of **events** and **event handlers**. Instead of writing plain actions that Flash will run regardless, you can enable a movie to trigger off scripts while it's running. By wiring up event handlers to button events, you can even give users a way to control what's going on in real time while the movie runs.

Don't worry if you didn't understand every single detail of what this chapter discussed; you've covered a lot of ground in a short time. Also, at this stage, a lot of the topics may seem a little abstract and theoretical because you don't have all the tools in place to try out all the possibilities this stuff opens up.

As long as you got all the exercises up and running, though, you should be able to cope just fine with what's coming up. From here onward, you'll be moving through the surrounding concepts in greater depth, but at a much more leisurely pace.





Chapter 2

MAKING PLANS

What we'll cover in this chapter:

- Planning your design project with the client in mind
- Building your ActionScript step by step to make your code easy to modify
- Structuring the flow of your ActionScript
- Introducing the Futuremedia case study

In this chapter, we'd like to show you an approach, as you're setting out on a design project, to defining the overall problem, splitting it into manageable sections, and structuring the ActionScript so that it gives you exactly what you need in a way that's easy to change if you need to.

We'll round off the chapter with a little reminder that web design isn't just a cold-coding process. We'll introduce you to Futuremedia, a website project that you'll work on as you progress through the book. By following Futuremedia's development, you'll learn how to plan usable interfaces and brew up cool vehicles to show off your cutting-edge ActionScript skills.

The best place to start is always the beginning, and that beginning is where you define what you need to do.

Whether you're a designer or a programmer at heart, there's one fundamental rule that you should remember: if it doesn't work on paper, it won't work when you code it.

Defining the problem

If this were a textbook on writing novels, this chapter would be all about defining your plot. Without plot, your characters would have no motive and your audience would quickly realize your book was going nowhere.

The same holds true for ActionScript coding. If you don't have a good idea of what you want to achieve, you're stuck. Most designers love to play around with ideas and end up with a toolbox of little demos and cool effects that they can put together later to build a full website. Even this organic way of working requires you to think about how your effects will fit together and what your aim actually is for a website. You should have a fair idea of direction *before* you start.

In real life you have a further complication that thwarts your ideas and clean, structured designs. That complication is the **client**. A good plan is a safety net that helps you guard against those little requests: "Can we just move everything half a pixel to the left?" or "No, we can't use that—didn't I tell you we wanted an advertising banner to go on top?" *("Uh, no you didn't . . .")* or the all-time classic "That's great, and I know we go online in two days, but here's a list of 35 minor changes that Bob and I would like."

The driving force of design should be managing the client's expectations and discussing (and finding) the client's real needs (as well as, of course, getting everything in writing so Bob and his boss don't feel like they can have a second redesign for free!). Formulating the project beforehand and getting agreement early on goes some way toward solving this. It also allows you to go back to the client and ask for costs if they start changing the project partway through (in the world of business, this is called a "variation on the contract," and you should always be in a position to be able to charge for this).

All the same, we'd be making false promises if we told you that, in a typical project, your design objectives wouldn't change at least once! This is something you just have to live with, and you should make sure that your design is defined well enough to let you go backward as well as forward, and branch out onto a new path if the client (or technical problems) force you to do so.

Your plans will help you through this, so you can point to them and say things like, “Well, if you want to make that change, this will have to go, this will have to move here, and that will go there. Is that OK?” without having to take the time to actually do it first to see what it looks like. This is a fact of web design, and though you may not experience it until you’ve finished a few paid jobs, you’ll be glad of your plans when you do.

> *As implied here, design is partly creative, but designers also have to build commercial and legal factors into the design process, and there are many issues behind a successful site other than just technical design and aesthetics. Although it’s beyond the scope of this book to discuss these issues in detail, you can have a look at the following links for more information on how to do some of the other things that fall under the general topic of “design” (or, more correctly, “managing a design project”).*

Writing a web-design contract The following are a number of sample standard contracts, ranging from the basic to the verbose. Note that some of them cover issues that you may not at first think of as important when writing the contract, such as browser compatibility and other gotchas that the client may come back to you with.

- `www.scottmanning.com/services/contract`: Scott Manning, Freelance Flash web designer. A typical basic contract. We wouldn’t recommend using this contract as is, although it forms a good base.
- `http://graphicdesign.about.com/library/contracts/uccontract1.htm`: This URL offers a good starting place from which to find several sample contracts.
- `www.emmtek.com/contract.asp`: EMMtek. A brief contract that covers most of the big issues without getting too technical.

Costing a website This is one of the thorniest issues for a new designer. Check out the following links. Also be aware that many web-design houses charge different rates for different parts of the job, and scripting (including ActionScript) can command some of the highest rates once you have a good portfolio. Don’t be shy!

- `www.webdevbiz.com/article.cfm?VarArtID=5`: WebDevBiz. This is a general article on costing a website. Although whether you should give estimates to some potential clients is open to debate, this article offers a good overview on how to cost a site (assuming that you’ve already decided on an hourly rate).
- `www.proposalkit.com`: Proposal Kit. This company sells a complete proposal and contract management collection of documents and templates for small-scale web developers. The kit contains templates for proposals, agreements, and related documents, and estimating and costing spreadsheets.

The other route to “per hour” costing is to charge per item created (typically per image, area of formatted text, menu, etc.). This isn’t usually done by small design houses/freelancers, so we won’t consider it. As a freelance ActionScripter, you would typically give only an estimate of hours and the rate you charge per hour.

Keep your ideas in a safer place than your head

So, you've talked to the client, and you've gathered a collection of ideas from things that you see every day and perhaps any assets (sound, music, video clips) that will play a part in your final site design. The next step is to tie down these ideas to form a concrete shape that shows you how you need to implement the site design.

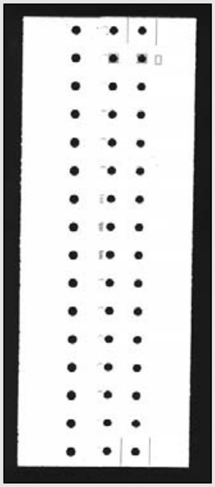
There are a number of ways you can do this:

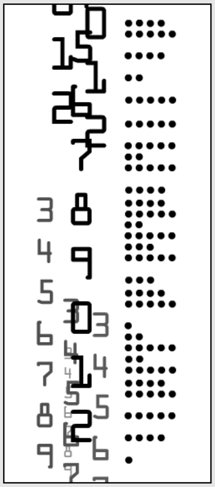
- Write down the essence of the site in plain English.
- Quickly code a basic sample site and discuss it with the client if possible.
- Storyboard the site, perhaps as a set of static Photoshop or Paint Shop Pro mock-ups, and present it for approval.

In fact, you'll most likely use a mixture of all the above. Many people find the storyboarding option the most useful, both in terms of communicating with the client and for their own use, to see how the site develops stage by stage.

Working in a field that requires us to think creatively doesn't mean that we're instantly creative. We all get good ideas, but most of us forget them with the passage of time. I now keep a hardbound book of lined paper for these transient ideas and a book of black card pages (black because it makes things much easier to scan into a computer) where I keep pictures that have caught my eye.

I have a big collection of images in my book now, from Polish Nightclub advertisements to stylized, textless instructions for inserting camera film told through icons. It's all great reference material. For example, I kept the edge of an old paycheck from a few years ago, one that was obviously printed by a big, fast, industrial computer printer. It has those funny sprocket holes in it, some print registration marks, and some optical character-recognition numbers. I would normally have put this straight in the trash can, but one day it struck me that it had a real made-by-computer feel, so I kept it. Several years later, I used it for the opening titles of a graphic novel I wrote and illustrated, which has an intelligent supercomputer as a main character.





You can see the progress from the grubby bit of paper on the left to the finished article on the right. You may think that it looks similar to the data animations from The Matrix, *but mine was based on a scrappy bit of paper and not an expensive computer animation!*

—Sham Bhangal

Storyboarding

Storyboarding is a good way to get a preliminary feel for what you want. It allows you to define navigation, visualization, and style early on in your site design. The term is a throwback to the film industry; it refers to the sequence of pictures drawn by cartoonists to plan how characters will move during an animation.

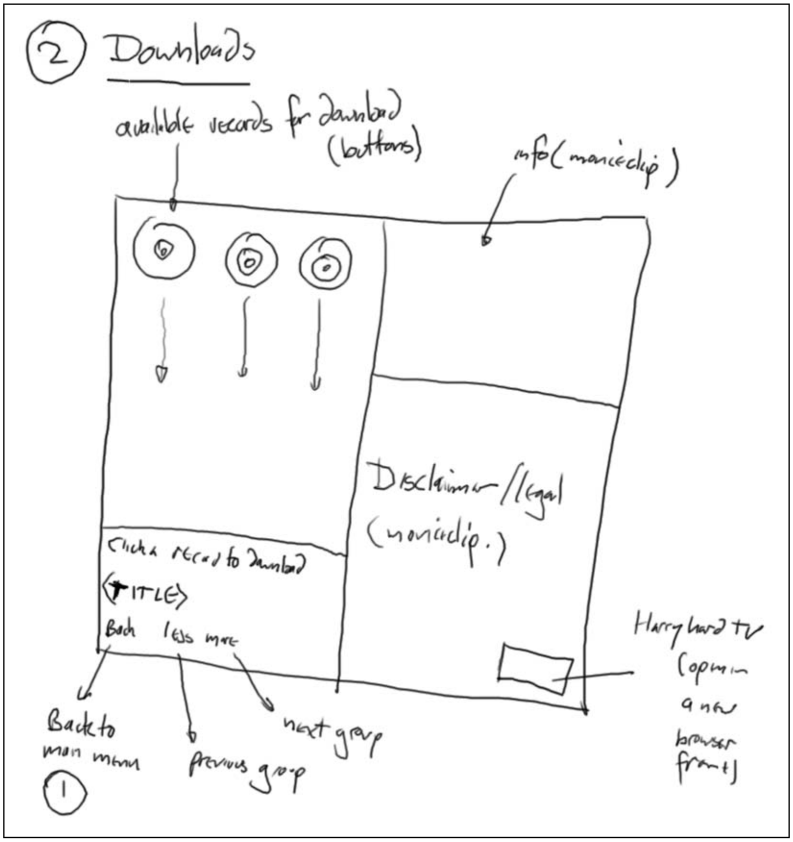
Some people may ask, "Why waste time creating static sketches when you could just as easily start drawing in Flash?" You may find that the little extra work you put into storyboarding really lifts a presentation and allows you to fully explore your creative ideas before you need to concentrate on developing and animating them in Flash.

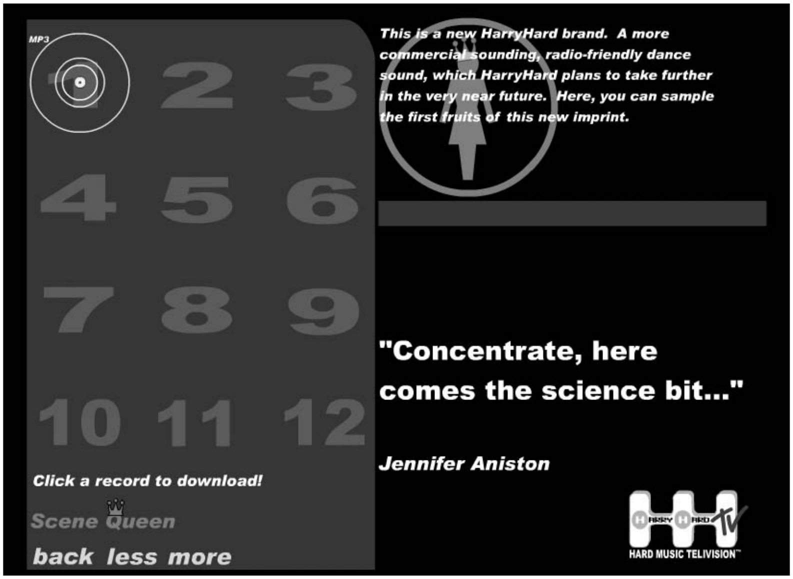
> *A rather well-kept secret is that it's easy to import a Photoshop PSD image into Flash, complete with layers. Export the PSD one layer at a time as PNG files from Photoshop, and you'll find that the Photoshop per-layer alpha channels are exported as well as the pixel information. Import the PNGs into Flash. If you place the PNGs in the correct order using Flash layers, you can easily re-create your Photoshop image in Flash, but this time around you can also animate it using either tweens (or, if you know what you're doing, ActionScript). Although this doesn't of course lead to a web-optimized Flash site, it is a quick and dirty way of transferring a Photoshop storyboard to Flash, for the purposes of showing the basic design to a prospective client.*

A storyboard can be whatever you want: a sketch on the back of a drink coaster just for your own use or a full-blown presentation of the site from beginning to end to show to the client. Some designers create complex storyboards showing a full website mocked up, with arrows and typed notes saying how everything will work. We recommend finding the middle ground. The client is paying you for the finished site and not the intermediate artwork, so make the storyboarding functional and don't go overboard—it isn't a finished product in its own right. Also, remember that a complete storyboard isn't something that you want to give to a client just so the client can give it to another designer! Keep the storyboard content that you give to a client to no more than a couple of pages.

On the following page is the kind of sketch that Sham begins work with (assume that the client hasn't requested to see it). This sketch is for part of a site he designed in which you can select from a number of different bands and download MP3 files (legal ones, we hasten to add).

Although the sketch probably took no more than ten minutes to draw, as you'll see, the final interface is very true to the original drawing. Having a sketchbook near the computer so you can draw a few rough ideas is always a good thing.


2 Downloads
available records for download (buttons)
info (movieclip)
Click a record to download
<TITLE>
Back less more
Disclaimer//legal (movieclip.)
Harry hard tv (open in a new browser frame)
Back to main menu 1
previous group
next group



This is a new HarryHard brand. A more commercial sounding, radio-friendly dance sound, which HarryHard plans to take further in the very near future. Here, you can sample the first fruits of this new imprint.

"Concentrate, here comes the science bit..."

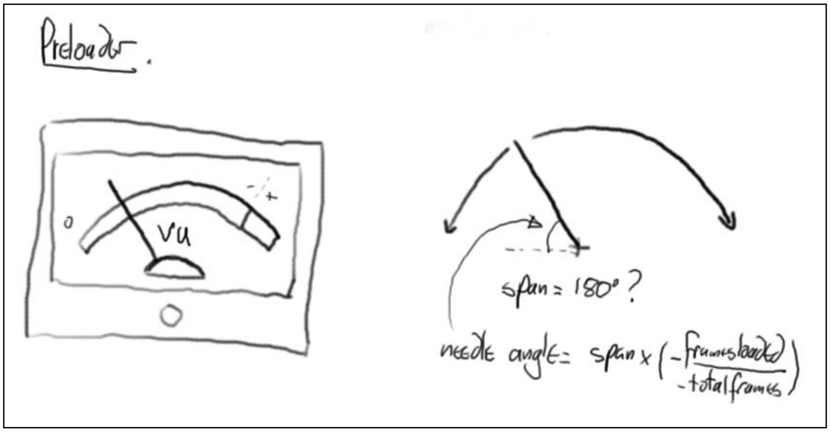
Jennifer Aniston

Click a record to download!

Scene Queen

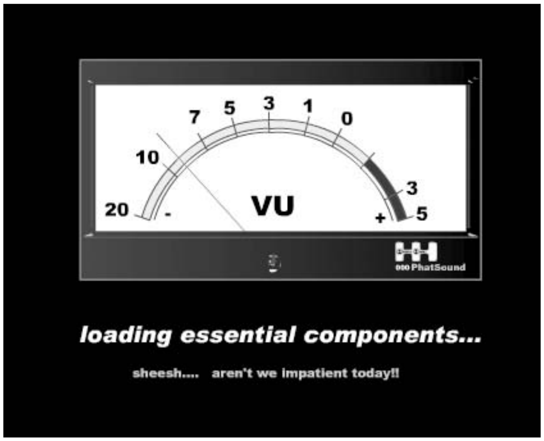
Trying to fit everything into Flash without an initial goal would have taken a lot longer, so this initial storyboarding saved time. Storyboarding also gives you an opportunity to start figuring out how to begin coding effects for particular areas of the site. On the music site, Sham had to deal with the fact that the user might choose to skip the intro before the essential components of the main site had downloaded. To cater to this, he needed something to show viewers how long they would have to wait and keep them amused for this short time.

He thought a VU meter would be quite a novel way of showing this, with the needle flickering up the meter to show the percentage loaded. As you can see from the sketch, in this part of the storyboarding, he's already starting to think about how the needle needs to be animated by ActionScript.



A volume unit (VU) meter is a common audio measuring device. It displays the average volume of a sound. Just the thing for a music website's preloader!

Here's the final work. Again, it's quite close to the initial ten-minute scrappy sketch.


20
10
7
5
3
1
0
3
5
-
+
VU
PhatSound
loading essential components...
sheesh.... aren't we impatient today!!

It's worth reiterating that Flash isn't necessarily the best environment for you to develop your initial ideas. Playing around within graphics packages like Photoshop or Paint Shop Pro, along with making your own sketches, can be a much richer way of working. Twenty minutes with pen and paper planning the interface and storyboarding how the ActionScript will work is time well spent.

Once you've sketched what you want to do, you can again save yourself a lot of time by staying away from your monitor just a little longer and not diving headfirst into the Actions panel. Take some time to plan in more detail how your code will be structured to fit with your designs.

Building your ActionScript

As you begin to think in ActionScript terms, you're always starting from the same point: you know how you want your designs to work. What you need to think about is the best way to get there. There are two main ways of breaking down programming problems: top down or bottom up.

A **top-down** design involves looking at the overall task and breaking the problem down into smaller and smaller chunks, whereas a **bottom-up** design means starting by looking at the basic building blocks, adapting them, and building upward toward the solution.

Thinking from the top down

The top-down method is called a **functional** method, because it looks at the functions that you have to carry out, or *what you have to do at every stage*, to perform the solution. You begin by looking at the aim in general terms or **high-level requirements**. Then you break up each of those general stages into separate steps that become easier and easier to manage. These are the individual **requirements** that you can deal with in turn on your way to building the complete package.

Sound strange? Let's say that you're looking at breaking down the high-level requirement of making a cup of tea with an electric kettle, a tea bag, milk, and water. If you take the top-down approach, you'll first define the top-level design statement, which is

Make a cup of tea and then break it down to its main stages.

1.0 Boil some water by filling the kettle with water and switching it on.

2.0 Put some sugar and a tea bag into a cup.

3.0 If the water in the kettle has boiled, pour some of the water into the cup.

4.0 After two minutes, remove the tea bag, add some milk, and stir.

Breaking the task down to this level is called the **first iteration**.

You'll then look at each of these tasks and break them down even further:

1.0 Boil some water by filling the kettle with water and switching it on.

- **1.1** Fill the kettle with water, two-thirds full.
- **1.2** Switch on the kettle.

2.0 Put some sugar and a tea bag into a cup.

2.1 Add one teaspoonful of sugar to the cup.

2.2 Add one tea bag to the cup.

3.0 If the water in the kettle has boiled, pour some of the water into the cup.

3.1 Wait until the kettle has boiled.

3.2 Pour some water from the kettle into the cup, until the cup is three-quarters full.

4.0 After two minutes, remove the tea bag, add some milk, and stir.

4.1 Wait two minutes.

4.2 Remove the tea bag from the cup.

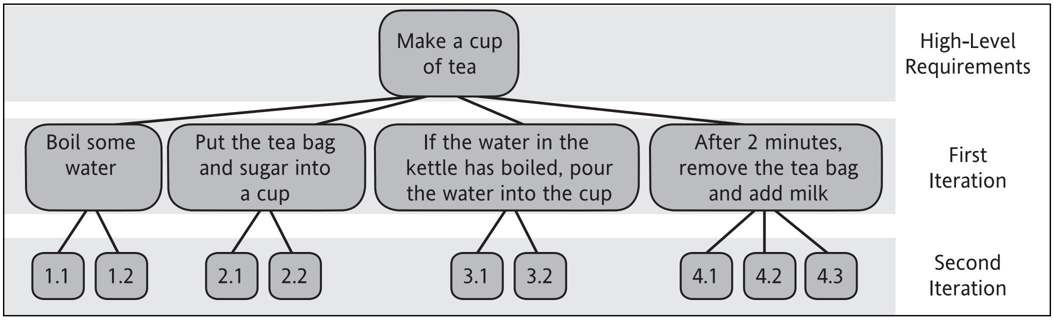
4.3 Add milk to the cup until the cup is seven-eighths full, and then stir.

This is the **second iteration**.

*You might be thinking that this looks suspiciously like a table of contents (or a **TOC**, as we call it in the publishing field). Well, that's exactly what it would become in a major project. This is the basis for a TOC for a top-down specification document for the making tea software project. Section 4.2 would (for example) introduce what the process is and perhaps provide further specifications about what removing a tea bag actually entails.*

Although this might sound trivial, section 4.2 would actually be a rather long section if a robot was making the tea. There would be technical drawings of the tea bag telling you how tightly the robot could grip the tea bag and where to do it from (you don't want it to burst in the tea), how the robot should find the tea bag in the tea, and all sorts of other clauses and subclauses.

Graphically, we can represent the top-down approach as a chart something like this:


Make a cup of tea
High-Level Requirements
Boil some water
Put the tea bag and sugar into a cup
If the water in the kettle has boiled, pour the water into the cup
After 2 minutes, remove the tea bag and add milk
First Iteration
1.1
1.2
2.1
2.2
3.1
3.2
4.1
4.2
4.3
Second Iteration

The second iteration would link to charts that further break down the problem.

> *A common question for the beginner is "When do I stop iterating?" Well, it's a bit of a black art, but a general answer would be when you've broken down the problem so that it's easy to see how you would code each of your iterations at the lowest current level in the chart.*

The top-down approach fits well when you're dealing with a tightly defined problem. A tight definition eliminates the need for too many iterations, or breaking the task down again and again, and makes each stage easy to change if necessary. For anything more complex, though, it throws up some pretty basic problems. The most important of these is that if the high-level requirement changes, that change must filter down through all the iterations, making you think about what you do at every stage all over again.

For example, if you prefer coffee, you'll have to start at the beginning all over again with a new top-level design statement:

Make a cup of coffee.

And you'll need to modify all the instructions at every stage below that, changing **tea bag** references to **spoonful of instant coffee**, taking out that **wait two minutes** stage (because now you don't need to wait for coffee to brew), and so on. You may think that this isn't such a big job in this domestic real-world example, but it would be very different if your top-level design statement reads like this:

Build a navigation computer system for a combat helicopter with satellite positioning systems, radar, and weapons management facilities.

You would have a bit more of a problem changing things around when the Army changed its mind about what it wanted its helicopter to do. You would need to look at all 10,000 requirements for the system and analyze the impact of the change on each requirement. This could take an extremely long time, and even longer if you have to prove that you did the analysis of the changes properly (of course, this ignores the fact that the Navy will invariably want something different from the seaborne version!).

Experience has taught us to go for the easier option when possible. Speaking of which . . .

Thinking from the bottom up

Making a change in a bottom-up design is nowhere near as difficult as in a top-down design. Taking the bottom-up approach involves looking at the problem, splitting it up into very general areas, and asking, "What are the basic building blocks here?" You look for basic structures within your problem and write general solutions for them before adding specific details to these general solutions to make them suit the problem that you're dealing with.

Going back to the tea example, the general building blocks of the problem are as follows:

- Moving things from one container (such as a sugar bowl) to another container (such as a cup)
- Measuring quantities
- Waiting for particular processes to complete (such as boiling the water in the kettle)

If you were tackling this in ActionScript, you'd begin to create actions based around these general building blocks.

You could start with a **moving** action, adding details of whichever containers and substances were involved:

```
move(fromContainer, toContainer, thing);
```

Then you could add a **measuring** action, in which you could display "how much" of "what substance" you want measured:

```
measure(quantity, substance);
```

Next, you could add **switching the kettle** action, using basic on/off commands:

```
kettle(on);
kettle(off);
```

A conditional structure allows you to do something (switch the kettle on or off) only when certain conditions are met (the water is boiling):

```
if (water boiling) {
  kettle(off);
}
```

You could then look at the problem and see how to build up to making a cup of tea using these structures. For example, to fill the kettle you'd want to use the move routine to fill it with water from the tap, so

- fromContainer is the tap.
- toContainer is the kettle.
- thing is the water.

The thing isn't just any amount of water, but two-thirds of the kettle, so you can express it as follows:

```
measure(water, twoThirdsOfTheKettle);
```

So the full statement to fill the kettle would be this:

```
move(tap, kettle, measure(water, twoThirdsOfTheKettle));
```

Taking it all in this way, the basic solution might look like this:

```
move(tap, kettle, measure(water, twoThirdsOfTheKettle));
kettle(on);
move(sugarbowl, cup, measure(sugar, oneTeaspoon));
move(teacaddy, cup, teabag);
repeat {
} until (event(kettleBoiled));
kettle(off);
move(kettle, cup, measure(water, measure(water, twoThirdsCup));
repeat {
} until (event(teaBrewed));
move(cup, trashBin, teabag);
move(milkContainer, cup, measure(milk, oneEighthCup));
stir(tea);
stop();
```

The two `repeat` statements will make you wait (repeatedly doing nothing) until the event you've specified has been detected.

This looks suspiciously like ActionScript already! You've lost the timings like "wait two minutes" from the previous top-down version. Why? Because it doesn't matter! You've moved away from the initial problem to the extent that what didn't really matter dropped out of the problem. You never even considered it. This method can save you a little work and is also useful when the main problem or design aim is likely to change frequently, which could happen more often than you think. You use a `move` routine to fill the kettle with water from the tap. Once you've written that general routine, you can use it over again, with just minor changes for several other actions, including

```
move(teacaddy, cup, teabag);
```

With the top-down solution, wanting coffee instead of tea caused you a major problem and you had to look again at the whole process. If you're faced with that problem here, the basic building blocks stay the same; you just have to change a few minor details again, and you can substantially reuse the existing code:

```
move(coffeejar, cup, coffeepowder);
```

That's much easier. Because the bottom-up design didn't just look at the top-level problem (making tea), you generalized the problem, or **abstracted** it. So, you don't care whether it's tea or coffee or crude oil—you can still use your basic `move()` and `measure()` building blocks again, and you won't have to change them when you do. Bottom-up design digs the foundations before building on top of them, whereas top-down just puts itself down in any old way. When the winds of change blow, you can guess which one stays standing longer.

This bottom-up design method can be much harder to understand than the top-down alternative because, as adults, we generally prefer to think **functionally**. Common sense says that bottom-up design shouldn't work, but it seems to work rather well. It really comes into its own with problems that can be reduced to a few types of building blocks, which means that it works particularly well with animation and graphic interfaces.

Graphic interfaces tend to have very few building blocks. For example, you can build a basic Flash site using only three building blocks: movie clips, buttons, and graphic symbols.

When you code in Flash, matters aren't usually as clear-cut as a straight choice between a top-down and a bottom-up approach. Most ActionScript structures will require a mix of the two. The aim of this section is to enable you to look at a problem and start to think about the best way to code it.

Next, let's take a look at a way of presenting a coding problem graphically that might make it easier when you come to writing the actual code.

Flowcharting

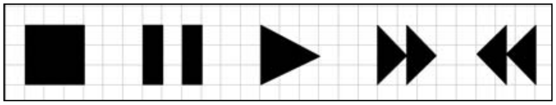
You've got an ActionScript problem—not necessarily with your whole site, but probably just the ActionScript for a single frame. You can't describe what you want to do to, so you might consider sketching the problem as a flowchart.

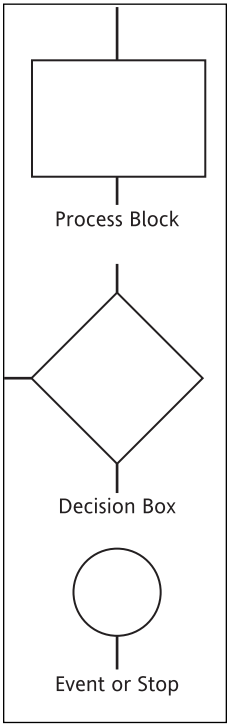
> *Thinking and looking before you jump straight into coding will change your mind-set from "keep coding until it works" to something altogether more elegant in thought and design.*

To build an ActionScript flowchart, you use three kinds of flow symbol:

- The **process block** represents a set of linear actions that occur one after the other.
- The **decision box** is like an intersection where you can continue down one of two paths, depending on whether the answer to your question is true or false (yes or no).
- The **event or stop** circle represents the event that starts or finishes a particular flowchart.

Let's look at how this would work if you wanted to build a menu of buttons that allows the user to control how a movie clip plays, just as if it were a videocassette in a video player. Once you create a set of buttons a bit like these, you'll want to attach some ActionScript to each of them.

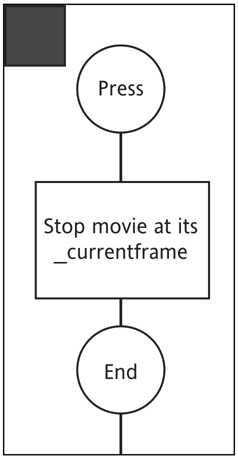




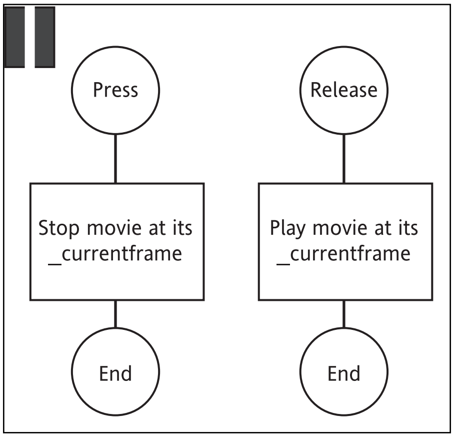
As you begin to plan, you'll make a note of some main points:

- You're interested in two main properties of the movie clip: `_currentframe` and `_totalframes`.
- The `_currentframe` property describes the frame that Flash is currently playing.
- The `_totalframes` property is the total number of frames within the movie clip (which is, of course, equal to the number of the last frame).
- The first frame is when `_currentframe == 1`.

With these points in mind, let's build some flowcharts.



The Stop button is fairly straightforward. When the user clicks it, you want the movie to stop. The flowchart shows that when a press event is detected on the Stop button, you need to have some ActionScript that stops the movie exactly where it is, at its `_currentframe`.

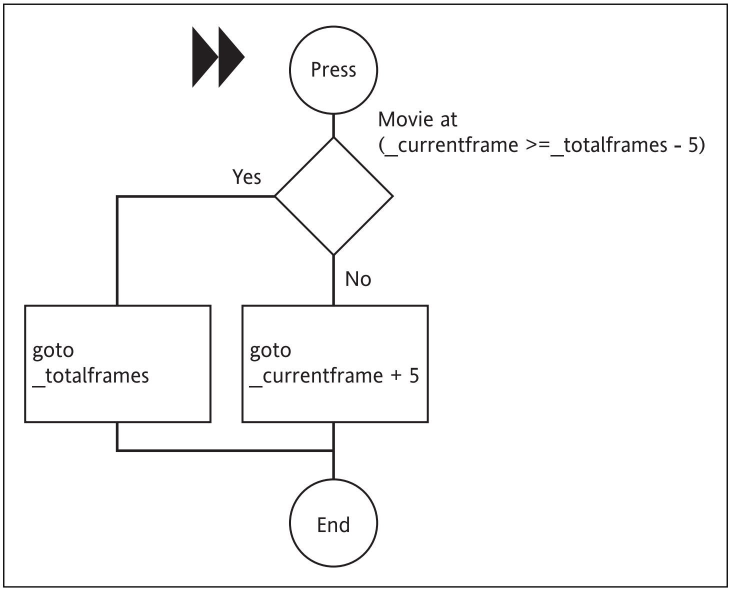

Press
Release
Stop movie at its _currentframe
Play movie at its _currentframe
End
End

The Pause button is slightly more complicated. When the user clicks it, you need the movie to stop just where it is, as if the Stop button had been clicked. But you also need to set the movie playing again when the Pause button is released. So, you have two events in the flowchart: Press and Release.

When the user clicks the FastForward button, you need the movie clip to advance at a much faster rate than normal. To make it move at, say, five times the normal rate, you ask it to go to `_currentframe + 5` every time the user clicks FastForward.

Houston, we have a problem. If you click FastForward when you're within five frames of the end of the movie, that means that you're asking the movie to go beyond the last frame. Unfortunately, Flash Player won't let you do this. It will probably come to rest in the last frame, but introducing an anomaly like this is never a good idea, as it might just break your movie. You should always try to ensure that your code is free from quirks like this that could introduce avoidable glitches. You therefore need to make sure that you don't ask your script to take you further than the end of the clip. So, your ActionScript needs to make a decision. If the `_currentframe` is within five frames of the last frame, you want the movie to go to the last frame. If it isn't, you want the movie to go to `_currentframe + 5` as normal.

If you draw a flowchart for this, you can see that you need to build in code to check whether `_currentframe` is greater than or equal to (>=) `_totalframes - 5` and create two routines to tell the movie clip what to do depending on the answer:



These diagrams show how you can help yourself to work out the logic that the final code must have, using a simple graphical tool. You could very quickly sketch these diagrams on a piece of scrap paper before you begin to build the ActionScript behind each button to give yourself some pretty useful visual clues.

Don’t go overboard, though. Flowcharts are useful, especially for designing a single event handler that consists of lots of decisions or loops, or both, but they do have their limitations. If you try to flowchart really long sections of ActionScript, you’ll quickly run out of paper. You may even discover that the flowcharts become more difficult to produce than the code itself!

It’s a good idea to use flowcharts to get a simplified bit of ActionScript going. Then, once it’s working, you can forget about flowcharts and slowly start building up the ActionScript itself. For example, with the FastForward button, the simple structure that you saw in the flowchart is fine as a beginning, but if things were left this way, users would find that they have to click the button over and over again to keep the movie playing FastForward.

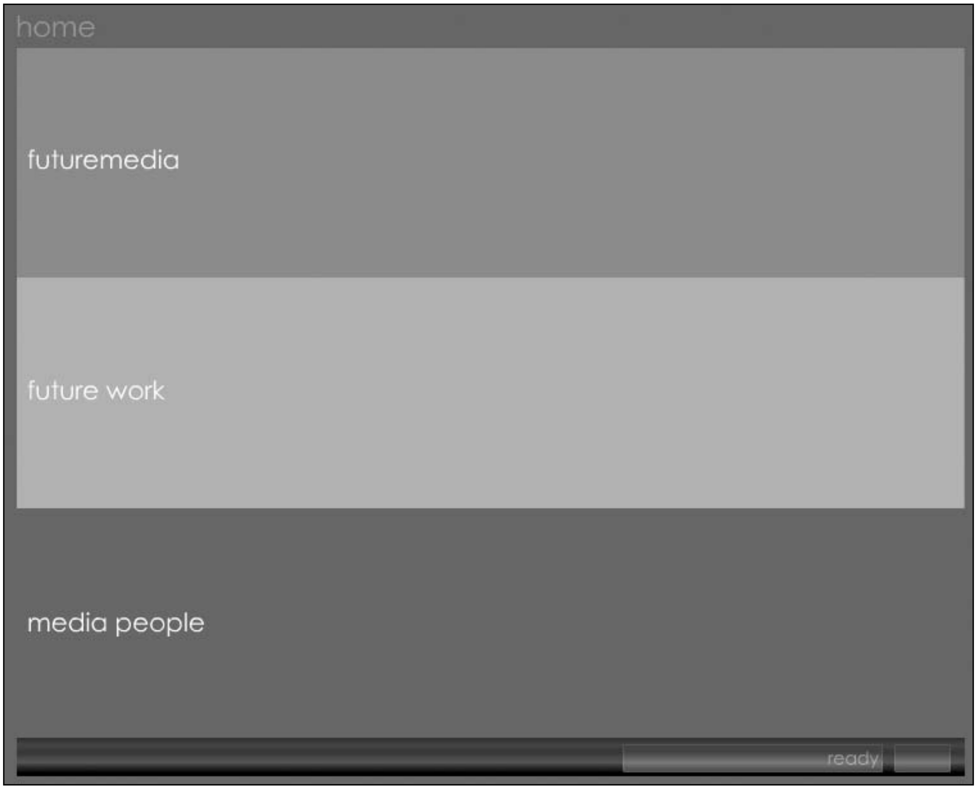
The solution for this would be to build a **toggle** mechanism into the button, something that makes it continuously alternate between two actions each time it’s clicked. The first time the user clicks it, it’s set to run on FastForward and continue running until it’s clicked again, when it’s set to stop.

Flowcharts are traditionally used when a solution is essentially linear, which means you do things in a defined order, such as calculating the average of 100 numbers. In user interface (UI) design, things work in the order specified by the user, which makes flowcharting difficult. You'll also find that flowcharts don't really tell you much about the way a Flash site will look and feel. Although most programming courses aimed at beginners sell flowcharts heavily, they're hardly used at all other than for setting up simple code structures.

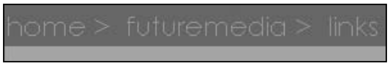
Let's put some of this theory into practice. It's time to introduce the book's case study.

Book project: Introducing the Futuremedia site

In the download for this chapter, you'll find `index.html` and five other files. Place all six files in the same folder (or on your desktop). Open `index.html` in a browser. As long as all the files are in the same place, you will see the Futuremedia site in the browser like this:



There are three areas to look at. The top left text indicates where you are now. As you move through the site, it keeps track of your current position in the site hierarchy.


home > futuremedia > links

At the bottom is a *visual* representation of the same thing. The pages change color as you move through the site, and this is reflected in the lower black strip, called the **back button strip**. It's sometimes also referred to as a **breadcrumb trail** because it's reminiscent of dropping breadcrumbs to mark a trail so that you can retrace your path.

The back button strip contains a small icon of the pages you have visited. Clicking any of the icons will take you back to a previous page with the same color scheme. Not only are the colors pretty, but also they're a critical part of the UI. The color coding is used to allow you to recognize the back path. There are no bells and whistles in a compact UI design. Everything has something to do with navigation, and put together, all these little cues and pointers (hopefully) create that elusive beast: a website that needs no instructions because its interface is totally stripped down and **intuitive**.

Finally (if you haven't worked it out yet), you can navigate the site simply by clicking the strip you want to go to next. The selected strip zooms and pans to fill the browser window, and if there are further pages to visit in the current navigation, it will split into strips again. It's one of those sites that's far more difficult to describe than to actually use. Click the navigation once or twice and it should all immediately (and intuitively) make sense. The navigation transitions are smooth, as are the color fades and transformations.

Right, that's what you'll design from scratch.

> *Of course, seeing the final site before you start is a bit of a cheat, but this is a learning exercise, so we can shortcut a little and show you the final piece at the beginning. In real life, this introduction would be replaced by the vision you have in your head. We hope you have the same vivid imagination as most web designers—you'll need it!*

The next question is *how do you start*?

The most important thing in any competition is to know your enemy, and the next important thing is to know how to beat them, and the best way to do that is divide and conquer. As you move through this project, you will

1. Work out what the problem actually is, so you know your enemy.
2. Subdivide the problem into easy-to-beat parts, so you can tackle each one on its own, which is something that gives you a much better fighting chance.

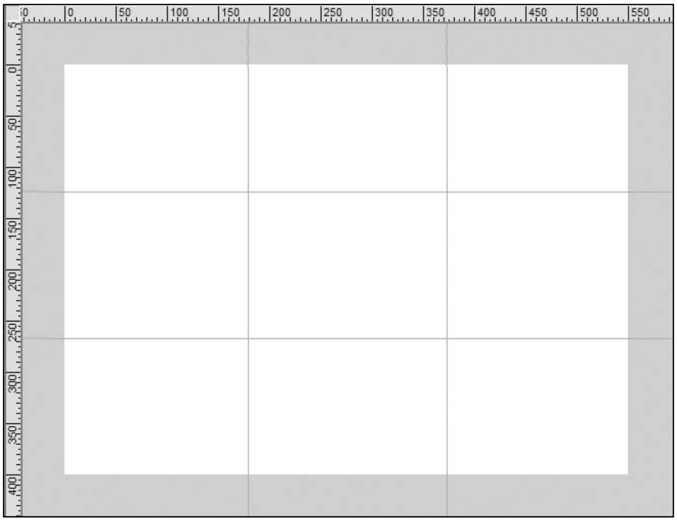
You can think it all through, but that tends to hide practicalities. Let's instead construct the method behind what the Futuremedia site does manually, but within Flash, to make a working sketch of what you need to achieve.

Deconstructing the problem

The following walk-through represents "edited highlights" taken from the design notes that show, *before* coding, exactly how the final site should operate.

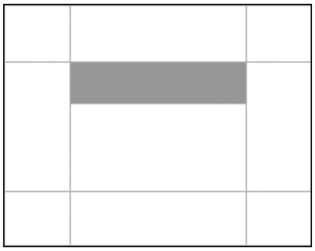
> *This part of the project will hopefully show how far you can move a coding task onward through the initial conceptual stages without even touching or thinking about the code. After the initial afterglow of the brainstorming session you went through to come up with an idea that may (or may not) be solvable with ActionScript, the following exercise is the sort of thing you have to do to prove to yourself that your design is actually feasible.*

1. Open a new Flash document. Select View ➤ Rulers to show the rulers. Click-hold-drag the horizontal and vertical rulers to drag out four ruler guidelines as shown here:

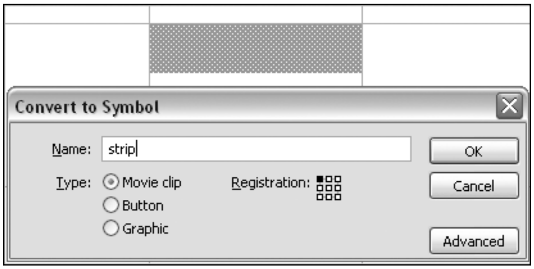


> *Copying the screenshots exactly isn't important in this exercise.*

2. The middle rectangle represents the viewable browser area. In this area, draw out a light gray rectangle as shown, making it cover about one-third of the inner rectangle. The rectangle doesn't need to have a stroke applied to it—a fill is sufficient.



3. This represents one of the three strips in the final site. Select it and convert it to a movie clip by pressing F8. Call it `strip`. For this exercise, the position of the registration point is unimportant, but when you come to build the final site, it will need to be top-left as shown in the following screenshot.



Name: strip

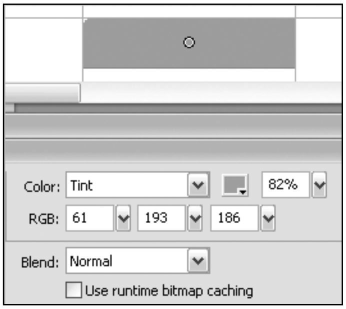
Type: ● Movie clip ○ Button ○ Graphic

Registration:

Why do you make it a movie clip? Why not leave it as it was? Well, the UI uses color changes as part of its operation, and only a movie clip can be made to change color dynamically. That is because `MovieClip` is a **class**, but if you want to see proof that you can't change the color of a raw graphic, here's the lowdown.

4. With the movie clip `strip` selected, open the Property inspector. The Color drop-down menu currently says None. Change it to Tint. By varying the RGB sliders that will now appear, you can vary the color of `strip`. If the strip doesn't change color, make sure that the percentage slider to the right of the color swatch isn't set to 0%.

   In the final site you'll have Flash varying the color via the `Color` class (a part of the ActionScript language), and you'll use it to create all those funky but subtle color transitions via code. For now, though, convince yourself that the `Color` class can't be used with a raw graphic by drawing another square, and this time don't make it into a movie clip. Notice that when you select it, the Property inspector doesn't show the Color drop-down menu. This is because the raw graphic doesn't have the properties that allow the `Color` class to work with it.



Color: Tint 82%

RGB: 61 193 186

Blend: Normal

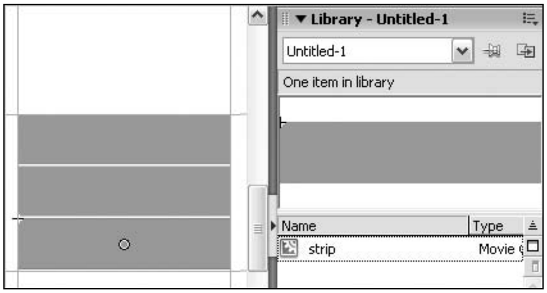
□ Use runtime bitmap caching

5. Delete the graphic so you end up with only the `strip` movie clip on the stage. Select the movie clip and return the Color drop-down menu on the Property inspector to None to bring the clip back to its original color.

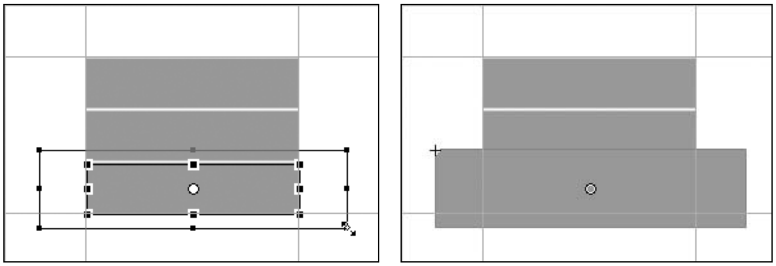
   You have your `strip` movie clip and are happy that there's a reason for it to be a movie clip and not anything else.

> *The ball, tennis racquet, and Starfleet SpaceCruiser in Chapter 1 were movie clips for the same reason: they had properties and an instance name, which Flash needs before it can control them with ActionScript. Flash just doesn't have the capability to control raw graphics in this way, and that's why ActionScript sites use movie clips all the time.*

6. Open the Library panel if it isn't already open (Ctrl+L/Cmd+L) and drag another two instances of the `strip` movie clip onto the stage to emulate the Futuremedia site's tricolor, as shown in the screenshot. If your strips are too big or too small, you can either scale them to fit or simply change the ruler lines around them.



7. So how do you scale this? Well, suppose you click the bottom strip. Flash will scale it and the other strips up as it zooms in to it. Let's try it manually with the Free Transform tool.



Whoops. There are two problems straight away.

One problem is that as the page starts to grow, it goes offscreen. The final site always keeps the selected strip onscreen as it grows. That's something your code will have to watch: *the selected strip must never be allowed to go offscreen*. Rather, it has to "push" the unselected strips offscreen to make room. The other problem is that only the selected strip does anything. The other strips just sit there, and the selected strip will eventually cover them. Now, these issues are actually the two critical problems of the whole site UI, because the site navigation relies on them working in some other way.

> *How do I know that this is the critical problem? Some of you might be thinking, "You know because you're a programmer, and I wouldn't get that insight because I'm not a programmer." Well, that does my ego a lot of good, but it isn't actually the truth. The truth is that I gained this "insight" by thinking about the design for a day. A whole freakin' day!*
>
> *—Sham Bhangal*

There's no magic point at which you become a master of Flash and just know. The thing about experienced programmers is that they have the courage and conviction to keep thinking about a problem until it's solved. As you progress you will, of course, realize that there are some routes that tend to work in Flash and some that tend not to (and you get a feel for what is a problem and what isn't), but all creative design is about taking leads and following them until one works, using judgment, creativity, and cunning in the process.

> *A lot of the process is knowing when something* won't *work rather than when it will work, and identifying it as such.*

Time and thought is the main resource, though.

Solving problem 1

Let's have a look at the "the two strips don't move when I scale the third" problem first. There are three possible solutions:

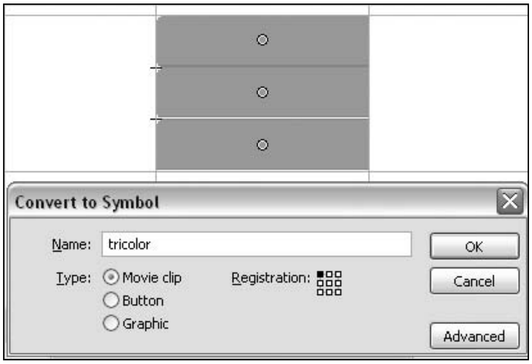
- Implement some sort of collision detection, so that you can detect when the expanding tricolor begins to overlap something else.
- Move all three strips proportionally so that they always keep the same relative distance apart and they'll never overlap.
- Cheat.

All the existing similar sites use the first or second solution, but we always have a soft spot for the third method when looking at a problem like this. We know this is one of those problems that's a **performance bottleneck**. Moving all three strips together in some controlled way will be a brain-wringer for Flash because it has a lot of things to do. It would be better if it all worked in some automatic way that required no coding.

Creating the tricolor movie clip

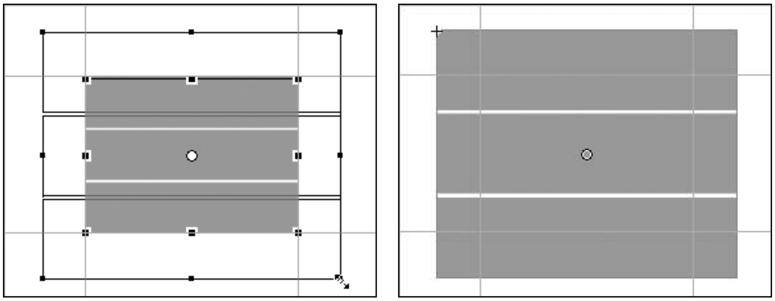
The following solution is *so* simple in hindsight, and it has the great advantage of requiring no coding at all. As a coder, you will learn to *love* such solutions, because they make your job all the easier!

1. Press undo (Ctrl+Z/Cmd+Z) until you get back to the three equally sized strips. Select each in turn so you have them all selected (hold down Shift while you select), and then press F8 to make them into a new movie clip.



Name: tricolor
Type: ● Movie clip ○ Button ○ Graphic
Registration:

2. Call the movie clip `tricolor`, because that’s what it is: a movie clip with three strips in it.
3. OK, now try to scale the bottom strip. See that? This time all the other strips scale as well *because they’re embedded in the same movie clip*!



Cool! So now the movie clips never overlap, and you don’t have to do anything highfalutin like collision detection (or, for that matter, write *any code at all*) to achieve this!

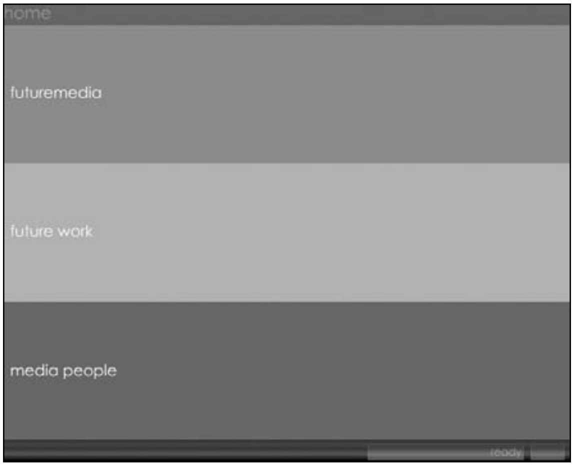
Solving problem 2

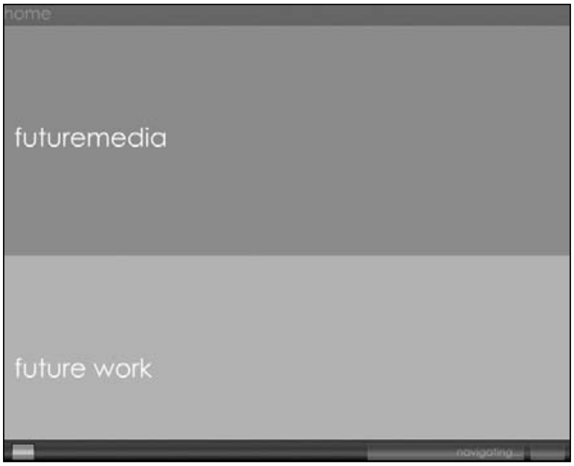
You want the bottom strip to stay onscreen as it scales (zooms) by moving it so that it never overlaps the ruler guidelines. This will keep it onscreen by **panning** so that it always stays on the visible screen. The best way to make something do *what you want* in programming, if the solution isn’t immediately obvious, is to look at *what you don’t want to do*.

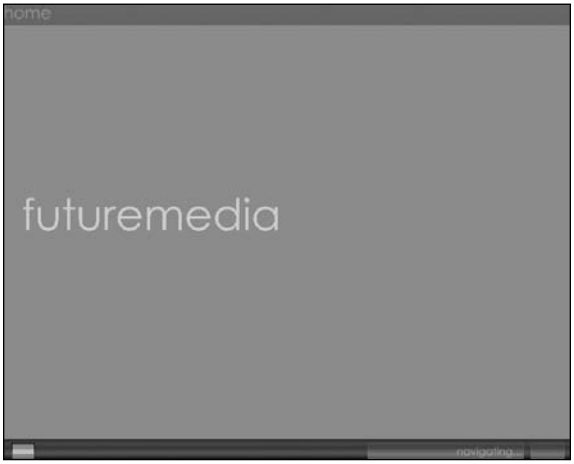
What makes the strip go offscreen? Well, you drag with the cursor and the tricolor moves off all four edges. If you look at the finished site, you’ll see that you don’t seem to care about the right edge—it’s the left, top, and bottom edges that concern you. A strip scales when you select it and grows off the right side of the screen. It doesn’t go offscreen on the other three edges as it grows. Once it has filled the screen, it stops.

So the final site does something different: it scales the tricolor, but at the same time moves the selected strip so that its top, left, and bottom-left corners never go offscreen.

You can model this by doing it yourself with a little Flash mock-up.

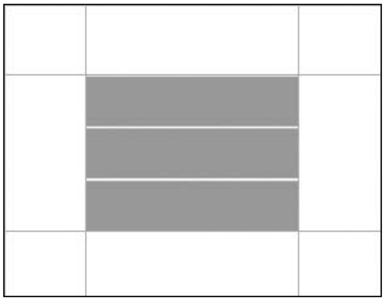

home
futuremedia
future work
media people
ready

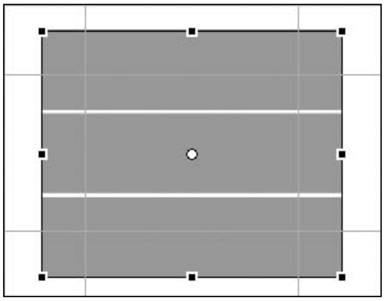

home
futuremedia
future work
navigating...


home
futuremedia
navigating...

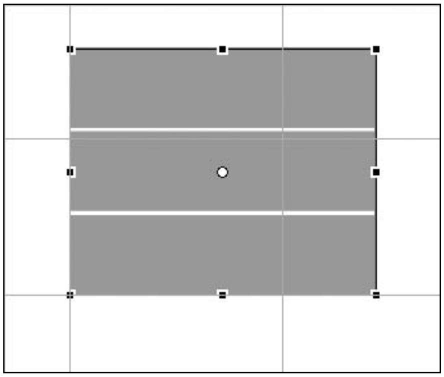
Modeling the tricolor transition

1. Starting with the original tricolor, suppose the bottom strip is clicked. You need this strip to scale so that it fills the screen, but as soon as you begin to scale it, you get the following situation, in which the whole tricolor goes offscreen on all four edges. Try this yourself, scaling the tricolor by a small amount.

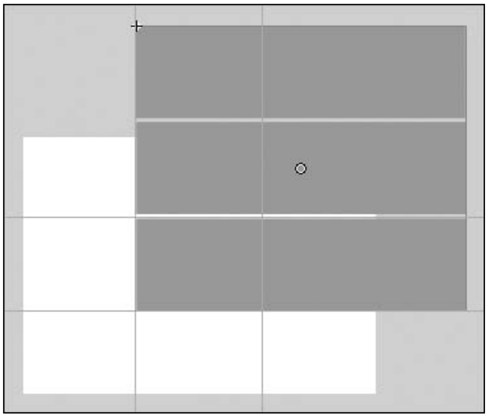




2. What you actually want to happen is this. Move the tricolor so that it occupies a position as shown. The left edge is kept at the screen edge, and the selected strip is prevented from going off the bottom.



3. If you keep doing this, you'll eventually end up with the strip hitting the top edge of the screen. Its color now fills the screen.



So that's how the final code does it! The three strips are actually a single movie clip that forms a tricolor. In addition to this, because the three strips are separate movie clips within the bigger tricolor, you can still access them individually to make color changes.

OK, so what happens next? Well, have a look at the final site design again.

You're halfway through a transition. You have one strip that now fills the screen, and two strips that are offscreen and can't be seen. Although the nonselected strips are offscreen, Flash still has to draw them, so you need to do something about them, or you have another performance bottleneck looming. You also need to change the one strip so it changes into the tricolor again. You're stuck!

> *Although you're seeing lots of problems in the walk-through here, this is a good thing, because you'll know about all the pitfalls up front when you come to code it. If you saw this new problem after you had already coded the zoom/pan functionality, you may have found that you would have had to discard the coding so far to overcome the new problem.*
>
> *A good programmer always plays about with the ideas before committing to coding because it uncovers the potential problems further on.*
>
> *A good motion-graphics programmer will do a bit more and take the animations manually through a dry run as you're doing here, because it uncovers issues that would be missed if you drew sketches or just thought about it. Doing it for real manually has several advantages, the main one being that it injects a healthy dose of achievable reality into your ideas!*

So you've solved problems 1 and 2, but you now have two new ones. That's not unusual, and not a worry. You know this because your overall solution has moved *forward*. You're now looking at new problems further on in the process.

Time to look at the options:

- Split each strip into another tricolor, so your original tricolor has further tricolors already embedded in it.
- Redraw the pages using the drawing API (you may have heard about this; you'll look at it in more detail in Chapter 11) so that they can be redrawn to look like three strips again . . . or something intense like that.
- Get Flash to somehow physically delete the extra offscreen strips.
- Cheat.

All the options are actually either inflexible or difficult, except our favored way forward. Guess which option we chose?

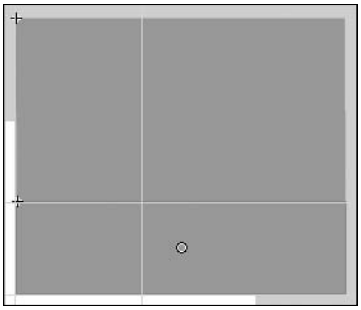
Solving problems 3 and 4

In all the best detective novels, the culprit is someone who was innocuously introduced early in the story. Nobody suspects the blind flower girl in the opening chapter, but that was the fatal mistake, for it was she who killed the countess (and she was only pretending to be blind).

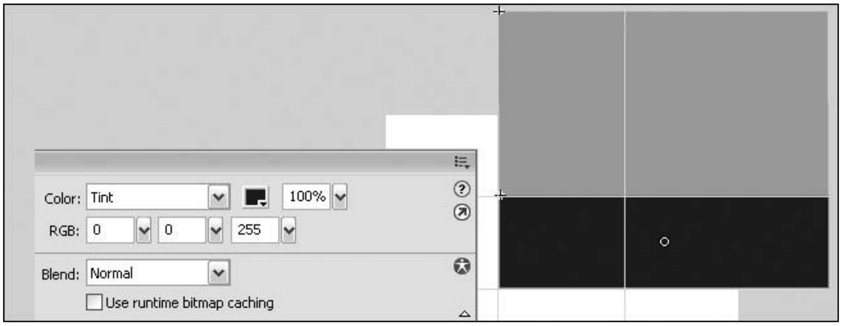
We introduced the solution early on: changing color. Use the by now oversized tricolor from the last part.

Adapting the tricolor

1. To see how this is done, you have to make this little test look more like the final site. Leave the tricolor in the same place and position it's currently in on the screen. You need to make sure there are no gaps between the strips within the tricolor. Edit the tricolor in place (double-click tricolor to enter this mode) to move all the strips so they just touch, as shown in the screenshot.



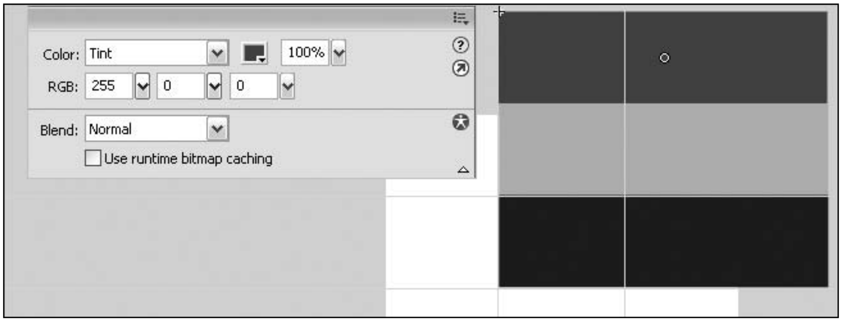
2. Change the color of the lowest `strip` instance via the Property inspector as you did before, to blue. Note that you still have to be inside the `tricolor` clip editing in place, or you'll tint all three strips.



Color: Tint 100%
RGB: 0 0 255
Blend: Normal
□ Use runtime bitmap caching

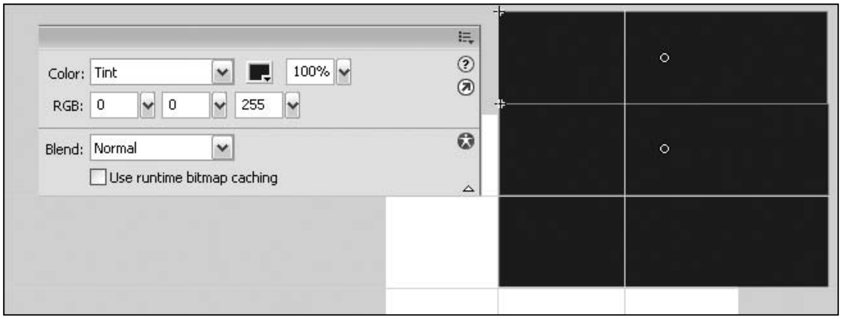
3. Give the other strips different colors. You can choose them yourself—the values aren't important as long as none of them is blue, for reasons that will soon become clear.

4. Let's recap. Only the blue strip is onscreen at the moment, and you want to somehow get rid of the other two strips that are offscreen. The cheat is that you don't get rid of them. You bring them back onscreen.



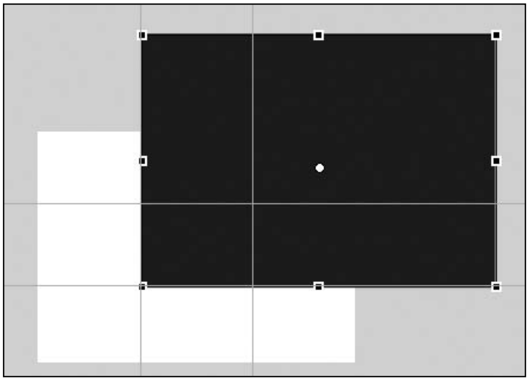
Color: Tint 100%
RGB: 255 0 0
Blend: Normal
□ Use runtime bitmap caching

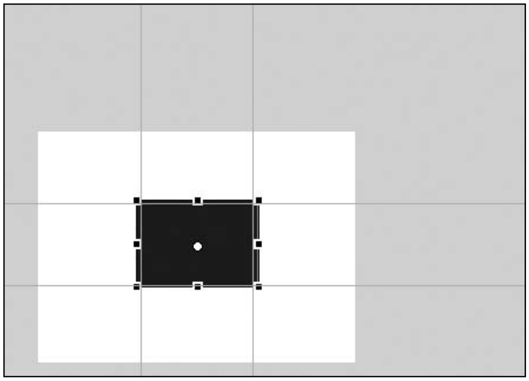
5. Change the color of all the nonblue strips to the same blue as the bottom strip.



Color: Tint 100%
RGB: 0 0 255
Blend: Normal
□ Use runtime bitmap caching

6. Now, here's the really cunning part. Get out of edit in place mode and *scale the tricolor* back to where it was at the start of the whole thing. Because the three strips are all the same color, the user won't notice this *and won't see the fact that you've brought the missing strips back onscreen,* because the user can't see what's going on offscreen.





All you do now is fade the blues toward the colors of the next page in the hierarchy, and users think they've gone to a new page because you've zoomed in to a strip, which then split into three new strips.

You've done no such thing! What you've actually done is zoomed out and then sneakily swapped back to the *same tricolor*, but with different colors. You don't constantly zoom in at all—it's a trick!

What's more, you're now almost at the starting point, with all three strips onscreen, so you can start the whole fake-zoom process again *using the same code*.

This is the difference between motion-graphics programmers (including programmers of PlayStation games and film special effects, as well as Flash web design) and other programmers—*we're always cheating*. The 3D in *Quake* isn't really 3D (it's on a 2D monitor screen, for a start), and the X-wing fighters in *Star Wars* are really 12-inch models or nothing but data in a computer. We cheat like heck. In fact, the bigger and subtler your cheats, the better you are as a motion-graphics programmer!

Of course, others call this "approximation," but we know different. Have another look at the Futuremedia site to get a feel for what is *really* happening when you click a strip and "zoom in" to a new page. Play about with it until you can see all the wires and mirrors and how the magician's assistant is really not sawed in half at all.

Parting shots

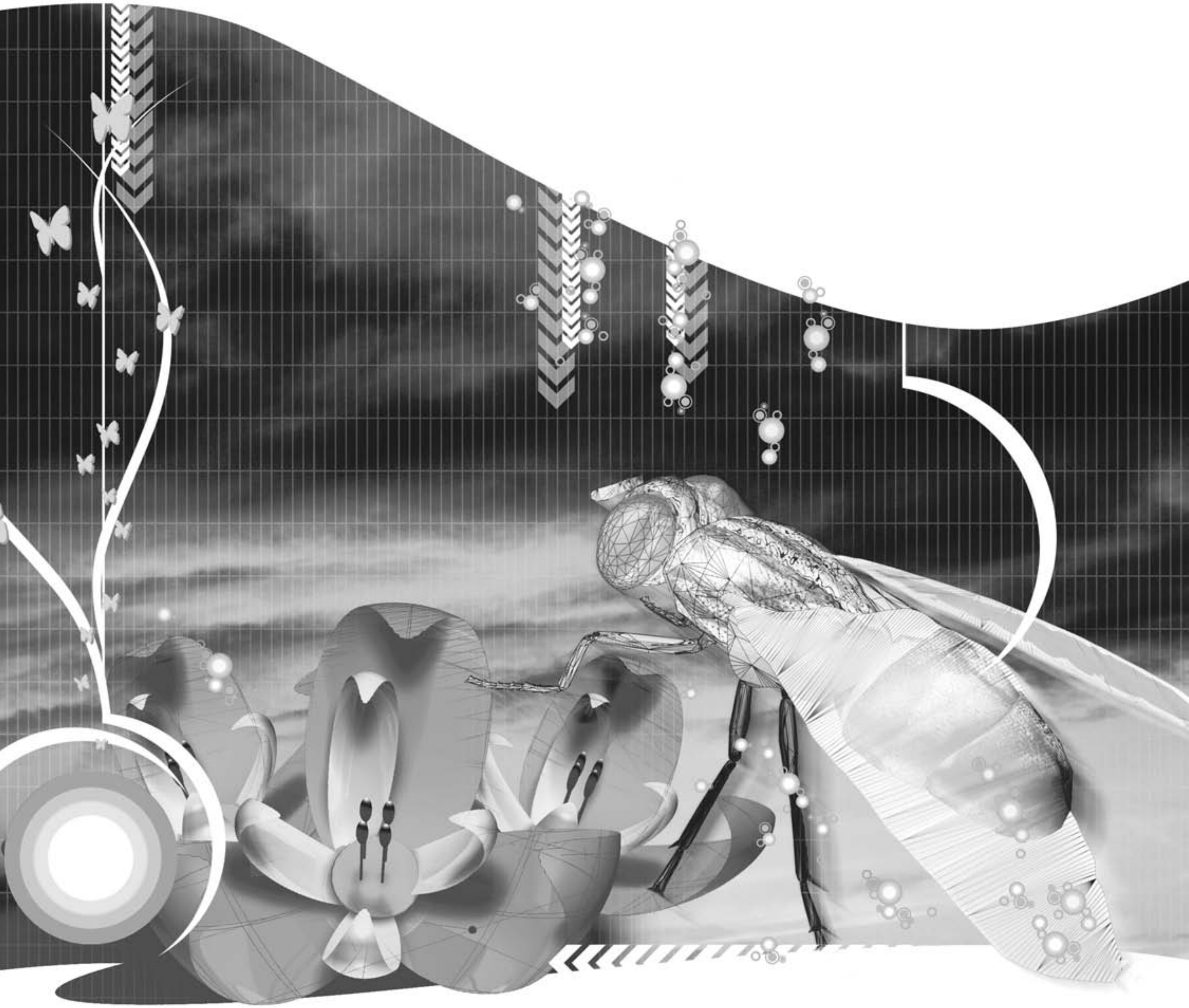
You've learned the first important rule in Flash motion graphics: *know what you want to code before you start*. You've addressed all the problems in the navigation you'll be using, and you've solved them in principle by running through the animations and transitions *manually*. You're much better off doing this by manually updating the graphics in real time than just dry-running it on paper, because motion graphics is, by definition, about *animation*. By doing it this way, you get some insight into what Flash has to do, and some shortcuts and approximations that you can make in your motion graphics, while still maintaining the desired effect.

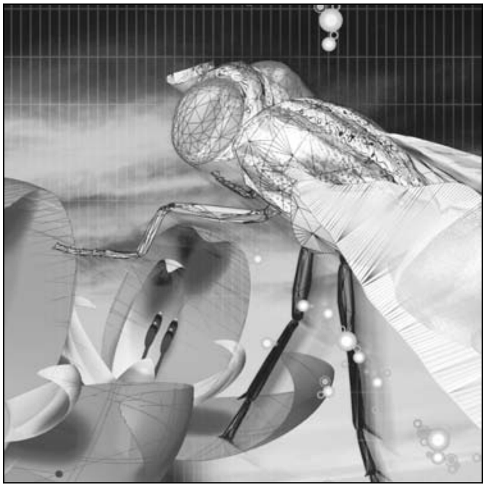
This is what creative ActionScript is all about. So far in the project, you haven't touched ActionScript at all, but we hope that already you're beginning to see something important: the *mind-set* that creates all that cool and impossible Flash stuff on the Web.

Best of all, Flash problem solving and coding isn't about math. It's about creating cool effects and stuff for other people to see, and we hope you're beginning to see the fun of it all—our problems aren't those of database programs or other serious applications. Flash programming is all about color, movement, and creating something that's cool and fun to interact with.

Summary

You've seen how planning is a crucial first step in designing a Flash site. It's particularly important when you're trying to develop something novel or new (such as the Futuremedia user interface) so that you can not only plan what you want to do, but also determine whether it's feasible!





Chapter 3

MOVIES THAT REMEMBER

What we'll cover in this chapter:

- Introducing variables as containers for storing information
- Understanding different types of variables: numeric, string, and Boolean
- Using expressions to generate values to be stored in variables
- Taking user interactivity beyond button clicking, using input and output text
- Using arrays to deal with complex, related information

So far, all our ActionScript has dealt with what's going on in the present, for example, what the *current* frame is or where the mouse pointer is *at this moment*. Of course, technology hasn't progressed far enough to let us peer into the future—not even with Flash! But looking at what's happened in the past isn't a problem. The only reason that all our movies so far have been stuck in the present is that they have no **memory**.

The advantages of having memory might not be obvious right away. After all, there's plenty you can do in Flash already, and you've seen how a little bit of ActionScript manages to raise the bar on that. What do you need memory for?

Once again, the answer has everything to do with interactivity. If you want to create a movie that makes users feel they're really involved with what's happening on the screen, you need to ensure there's a bit of continuity to the experience. For example, take the Starfleet SpaceCruiser game you looked at in Chapter 1 and think about how much less fun it would be to play if you couldn't give it any kind of scoring system.

> *In short, there's nothing wrong with movies that have no memory, just as there's nothing wrong with movies that don't use ActionScript at all. It's just that there's a whole lot more you can do with it than without it!*

Having memory means having a place to store **information**, or **data**. Once Flash has stored a piece of data, it can keep the data there for as long as it likes and call up the data whenever required. Flash can still *respond instantly* to whatever's going on (as you saw in the last chapter), but once it has memory, it can choose *not to*, and respond whenever it chooses.

Even more important, once Flash has somewhere to *store* data, users can send it much more interesting messages (like text and numbers) than they'd ever manage by clicking a button.

Virtually all your ActionScript-enabled movies are going to use some kind of data, and it's fairly normal to find that you'll want to change it from one moment to the next. For precisely that reason, the ActionScript "containers" you use for holding data are designed so that it's extremely easy to vary their contents—that's why they're called **variables**.

Here's an official-sounding definition:

> *A variable is a named memory location for holding data that's prone to frequent change and required to be accessible rapidly.*

In the first part of this chapter, we'll take a look at what this definition really means and see how Flash uses variables to keep track of what's going on, to make decisions and predictions, and to perform calculations.

Does it sound like you're in for some heavy math? No, and don't let the idea of math put you off! Variables don't need to be used to contain things like the speed of light squared—they can just as easily be used to specify how high you want your cartoon Flash character to jump. It's not math; it's just a powerful way to get Flash to do the fun things you want it to.

So, without further ado . . .

Introducing variables

A **variable** is a named "container" that you can use to store data that you need to use several times (and possibly change) during the course of a movie. Whenever you need to use the data, you give Flash the variable name, and Flash will look in the right place and retrieve it for you. If you've never wrangled with code before, this concept may be confusing, so let's consider it in nonprogramming terms.

Your wallet is designed specifically to hold money (hopefully you have some to put in it!). At any given time, the wallet will contain a particular amount of money, which will increase and decrease as you visit ATM machines and stores full of cool and tempting goodies. The wallet itself probably isn't worth much; it's the money inside that defines its value. Whatever you paid for it, your local superstore isn't going to accept an empty wallet as payment for produce!

To put it another way, the wallet is a container that you recognize as holding your money. When you hear the word "wallet," you know that what you'll find inside is money—never socks or frozen pizza. You also know that the amount of money in the wallet won't stay the same for long.

Of course, you might use other containers to store money, too. You might have a tin in the kitchen to store cash to pay for coffee, tea, and chocolate cookies. You might have a piggy bank or a bottle into which you throw all your small change throughout the year. (You might even spend that big bottle of money on a small bottle of something else at Christmas . . .) You have different containers because you need separate places to store money that's set aside for different purposes.

So you have three different storage places, each holding separate quantities of money that fluctuate constantly:

- Your **wallet**, containing your day-to-day money
- Your **tin**, containing money for tea
- Your **bottle**, containing your Christmas beer money

You can think of these storage places as being your "variables": each contains a different value that may change at any time. The first variable is `wallet`, and the value it contains is the amount of money that you have to spend on everyday items. You also have a `tin` variable that holds tea money, and a `bottle` variable that holds beer money.

An important aspect of this high-level financial management is that you keep money for the right purpose in the right variable. Imagine the tragedy of confusing your `tin` variable with your `bottle` variable, so that at the end of the year you have enough cash to buy tea for the whole of China, but only $2.50 for a glass of beer on Christmas Eve!

You're not very likely to confuse physical items like a tin, a bottle, and a wallet, but you don't have that luxury when you're programming. To ward off potential tragedy, you'll have to stick labels on the front of the containers, or name your variables so that you know what each one holds.

Even with labels, you can still get lost. You can have containers that have the same name, but in different places. For example, you might have a bottle in the kitchen labeled "Christmas," for buying a Christmas drink or two, and a piggy bank in the living room labeled "Christmas" that contains savings for gifts for your friends and family. In the same way that you can have containers for Christmas money in two rooms, you can have variables on different timelines. Obviously, you therefore have to know where each of your containers is as well as the name of the container. If not, your friends might be a bit annoyed when they see you at Christmas and you're very drunk, having spent all the money meant for their gifts on beer!

The location of a variable is called its ***scope****, and it can sometimes be as important as the variable's name. The scope is basically the area that the variable exists in (or where it's visible to the code). You'll look at scope closely in later chapters.*

Values and types

In the real world, you're surrounded by different kinds of storage spaces designed to hold different things:

- Your wallet is designed to store money.
- Your refrigerator is designed to store food.
- Your closet is designed to store clothes.

In ActionScript, too, there are different kinds of values that you might want to stow away for later use. In fact, there are three such **value types**:

- **Strings**, which are sequences of letters and numbers (plus a few other characters) such as *Kristian*, *David*, *Hello*, or *a1b2c3*. String values can be any length, from a single letter to a whole sentence, and they're especially useful for storing input from visitors to your site.
- **Numbers**, such as 27, or –15, or 3.142. As we'll discuss later, putting numbers in variables can be useful for keeping track of the positions of things in your movies and for controlling how your ActionScript programs behave.
- **Booleans**, which are "true" or "false" values. These can be useful for making decisions.

Some programming languages can be very fussy when it comes to storing different types of data, forcing you to work with variables that can store only one kind of value. If this were true in the real world, you'd discover it was physically impossible to hide chocolate at the back of the bedroom closet or put your jeans in the refrigerator. And what a cruel world that would be!

ActionScript lets you decide whether or not you want it to be fussy. By default it's an **untyped language**, which means that you don't have to specify what sorts of values a variable will hold when you create it. If you use this default, ActionScript variables can hold different kinds of data at different times. In most cases, Flash can work out for itself what the type is. Flash isn't infallible, though, so this flexibility sometimes means you need to exercise a little strictness yourself, which in turn means that there's the potential for *you* to get it wrong and fool Flash into making mistakes.

Sometimes it's important to be strict with Flash and tell it what sort of data to accept—like telling your kids not to leave their shoes in the refrigerator. When you use Flash this way, you're creating code that's **typed** (also called **strictly typed**). Writing code this way forces you to always consider the type of each variable when you use it, and Flash will warn you when you get it wrong. Although code written this way takes longer to produce, it's less likely to be wrong.

You'll look at some examples of untyped and typed code later on in this chapter, although for much of the book you'll be using the one that creates the more robust code: typed.

> *Always try to work with the value type that's most appropriate to the particular problem you're solving. For example, we'd all like to be able to converse with machines in a language like the one we use to talk to each other. Strings are useful here, because they represent a friendly way for Flash to communicate with users. On the other hand, strings are no good for defining a complex calculation; for this, you'd want to use numbers.*

Creating variables and using them with literals and expressions

Shortly, you'll look at some examples that will start to demonstrate just how useful variables can be. When you get there, you'll find that Flash gives you quite a lot of help, but it's often nice to know what to expect.

The easiest way to store a value in a variable is to use an equal sign (=) like this:

```
variable = value;
```

Whatever value (whether it's a string, a number, or a Boolean) you specify on the right gets stored in the variable you've named on the left.

Note that = here isn't the same as "equal" in math. You are *not* really equating the left side to the right side (although if you assume that you are, nothing untoward will happen to convince you otherwise); rather, you're **storing** the `value` in a container (or location) called `variable`. Programmers sometimes call this process **assigning**—you assign the `value` to the `variable`**.**

Naming variables

In Flash, you can name a variable however you like, *so long as Flash isn't likely to confuse it with something else*. Built into Flash are numerous names and symbols that are important to its correct operation, so you can't use them for your own purposes. These are called **reserved names**, and they include the following:

- **Actions** and other parts of ActionScript that might be mistaken as *part of an action*, including `{}`, `;`, and `()`. Examples of reserved names that cannot be used as variable names are `break`, `call`, `case`, `continue`, `default`, `delete`, `do`, `else`, `for`, `function`, `if`, `in`, `new`, `on`, `return`, `set`, `switch`, `this`, `typeof`, `var`, `void`, `while`, and `with`.

  Of course, actions themselves can't be variables. Basically, if it shows up in dark blue when you type it, then it's a reserved word and you can't use it. Obviously, don't start your variable name with `//`, because then it just becomes a comment!

- **Spaces** in a variable name will confuse Flash into thinking the name has finished before it actually has. For example, `kristian besley = "author"` will look to Flash like a variable called `kristian` that's followed by a string of random gibberish.
- Variable names that **begin with numbers** (such as `6pic`) won't work either. Use `pic6` instead. `p6ic` is also OK for Flash (although it looks a little cryptic to us!).
- **Operators** such as +, -, *, and / should be avoided. Otherwise, Flash may try to treat your variable as a sum to occur between two variables.
- The period character **.** is actually another operator (though you may not have realized it), so you should steer clear of using it in your variable names as well. However, as you'll see shortly, this character plays an important role in how you **locate** variables.

Also, Flash is case sensitive, so although the variable names `mycat`, `MyCat`, `MYCAT`, `MY_CAT`, and `myCat` are all possible, it's a good idea to pick a naming system; otherwise, you'll become very stuck very quickly. For example, if you name a variable as `myCat` and later use `mycat`, Flash will treat them as two separate variables without telling you anything is wrong. You will, of course, know something is wrong (your code most likely won't work), but it will be difficult to see what has gone wrong—the difference between "c" and "C" isn't much when you're looking through a 400-line script!

Flash has only been case sensitive since the previous version (MX 2004), so you might find that many older Flash files are dysfunctional if they were programmed without any consideration to the casing of variables. This might be the case with scripts you might have downloaded from a Flash resource, so beware.

The standard naming strategy used in the Flash community—and, for that matter, in most scripting languages (oh, and in this book too)—is as follows:

- Start normal variables with a lowercase letter everywhere you start a new word and an uppercase letter wherever you would normally have a space. Your chosen standard variable name would thus be `myCat`. As explained in Chapter 1, this format is known as **camelCase**, so named after the humps it creates because of the change in case (who says programmers don't have imagination?).
- Create all variables that you don't want to change in value once they've been assigned (constants) in `UPPERCASE`. When you want to add a space to a constant, add a _ (underscore) instead. So, if you want a variable that holds the name of your cat (which isn't likely to change), you use the variable name `MY_CAT`.
- Don't use any of the other available permutations of the possible variable names. Flash won't raise an error if you do; it's just that you're more likely to make mistakes as you write your code if you mix naming styles.

With these provisions in mind, some example valid and recommended variable names are as follows:

- `cow`
- `cowPoke`
- `menu6`
- `label7`
- `off`

Here are some examples of invalid and not recommended names:

- on, because Flash will expect it to be part of an onEvent action.
- label 7c, because Flash will see 7c as garbage due to the space before it.
- 6menu, because it starts with a number.
- my-toys, because it contains a minus operator, so Flash will assume you actually want the difference between the two variables my and toys.
- play, because ActionScript has an action called play().
- TOP_menu, because it doesn't fit into this variable naming system. Although Flash won't complain, you're likely to get the variable name wrong at least once in any long piece of code, and it's tough to find the mistake.

There's one slight complication: suffixes. Rather like your operating system uses file extensions such as .exe, .jpg, and .zip so that it (for example) knows that flash.exe is an executable file, Flash has a system of suffixes that let the Actions panel know what the type of each variable is.

For a string variable, use the suffix _str. For Booleans or numbers, leave them as is. A good string variable name would be myVariable_str rather than myVariable, which by implication would be either a number or a Boolean.

Creating variables

You have two ways of creating variables in Flash. First, you can simply start using them, and if Flash sees a variable name it doesn't recognize, it will create a new one to suit. The following line will create a new variable called myVariable and assign it the value 10:

```
myVariable = 10;
```

Second, you can define a variable with the var keyword. Both of the following are acceptable (and do the same thing):

```
var myVariable;
myVariable = 10;
```

or

```
var myVariable = 10;
```

Now you might be thinking, "Heck, both of these are longer than the myVariable = 10; line. Why go the extra mile here?" Well, using var allows you to create *much* more structured code. In the myVariable = 10; line, it isn't obvious that you're creating a new variable, so it may be hard for you to see what's going on six months down the line. Using var is a signpost to you that says "I'm creating a new variable here."

Another good reason for using var *is that it allows you to more fully define the variable, rather than leaving Flash to its own devices and just taking what it offers you. Using* var *enables you to do clever things with scope and type, as you'll see soon.*

In this book, we'll always define a new variable with var because it's a good programming habit to get into.

Once you've obeyed these rules to keep Flash happy, the important thing is that your variable names keep you happy, too. You don't want to spend time figuring out what types they contain or what values they're meant to hold, so be as efficient as you can. Make sure that the names say something about what the variables hold.

For example, if you have three variables holding your three amounts of money, you could name them dayMoney, teaMoney, and beerMoney. This will really make debugging a lot easier. As you become more experienced, you may find there are times when you need to use variables of the same name in different places. As long as you're clear on the reasons why you need to do that, it's fine.

The easiest way to specify what you want to store is to use a **literal value**. Let's see what that means.

Using literal values

Imagine you want an ActionScript variable called myName_str. If Kristian wanted to store his own name in this variable, he might do it like this:

```
var myName_str;
myName_str = "Kristian";
```

Here, "Kristian" is a **string literal**. The string value that you want to store in myName_str is **literally** what appears between those two quotation marks. Why do you need the quotes? Simply because Flash is too dumb to know which part of your code is the string literal and which part is code. Code, after all, is simply text. Quotes always enclose string literals so Flash can make the differentiation.

It's important to include the quotation marks in a string literal. Without the quotes, Flash will think that you're referring to another variable. Say you have this:

```
var myName_str;
myName_str = Kristian;
```

Instead of setting the value of myName_str to the word "Kristian," Flash will go looking for a *variable called* Kristian because it assumes the "Kristian" part of the line is *code* rather than a literal value. If such a variable existed, it might contain any number of different things—probably not what you want, though! You always need to use the right syntax to tell Flash what you want it to do, and for literal string values, that means using quotes.

In case you're wondering, if you do tell Flash by mistake to set a variable to be another variable that doesn't exist, it will set that variable to a special value called undefined. *This isn't the same as holding the string literal "undefined," but it tells Flash that there's no data whatsoever in there.* undefined *isn't the same as 0, because the latter is a definite value.* undefined *is less information than even that—you just don't know what the value is!*

Similarly, you could have another variable called myAge. To create this variable and store a value in it, you might use a **numeric literal**, like so:

```
var myAge;
myAge = 28;
```

After these two lines, the variables myName_str and myAge can be used in your code as direct replacements for the values they contain. Although your code can't change the literal values Kristian and 28, it *can* alter the values of the variables myName_str and myAge—they can be made to contain something else later. That might not sound like much now, but you'll soon see just how much versatility this can provide.

> *Remember that variables live up to their name: storing a value is no bar to storing a different value in the same variable later on. Variable* x *may start out holding the number 10, but you could make it hold a new value just by writing* x = 20;.

Using expressions

The next way of generating values to be stored in variables is through the use of **expressions**. As you'll see in the exercises, this is where programming starts to look a bit like math, because expressions are often sequences of literals and variables linked together by mathematical symbols. For example, you might have something like this:

```
var myAge, myResult;
myAge = 28;
myResult = myAge - 10;
```

More than one variable can be declared at the same time by separating them with the comma character as just shown.

Flash would calculate the mathematical expression 28 – 10 to reach the answer 18 and store this new value in myResult (or *assign it to* myResult).

It's when you start to use expressions like this that you have to think carefully about the types of data being stored by your variables. The price you pay for the ease of using your untyped variables is that it's **your responsibility** to keep track of what they're holding. If you get it wrong, you run the risk of using a value in the wrong place and producing incorrect results.

Think back to your high school algebra. You can plan an equation like $a + b = c$ and be sure that it will work even before you know what the exact values of a and b are. Whatever happens, they're both numbers, and you know that two numbers added together will give you another number. Easy.

Programming is only slightly more complex. Variables don't always hold numbers; as you know, they can contain other things such as letters and words. You have to be aware of the types of data and make sure that you don't try to mix two different types together.

If variable a = 2 and variable b = 3, you could add a and b to get a value for c that makes sense to a math-savvy person, because 2 + 3 = 5 (well, except to a physicist, who might tell you that it equals 6 because you're using a heavy 2).

You can also use expressions with strings. If variable a = "mad" and variable b = "house", you can add a and b to get a value for c that makes sense: "mad" + "house" = "madhouse".

When you mix up variables, though, you can get some unexpected results. For example, if variable a = "mad" and variable b = 3, and you add them together, the value for c is unlikely to make sense, because "mad" + 3 = "mad3".

To make variables work for us, we have to manipulate types separately and not mix them together. As a rule, if you input all numbers, then you can expect the result to be a number as well. If you input a string, then you can expect the result to be a string. If you mix a number and a string, then the result will be a string, too. If this doesn't happen the way you expect it to, it's important that you know why, because this can be one of the most subtle and difficult errors to find if you aren't expecting it (and you'll see a way around this almost immediately when you look at strict typing). One simple error is assuming that because a string consists of only a number, then Flash will treat it as a number, but this isn't the case. For example, if you have the following expression:

```
myAge = 33 + "3";
```

you might expect the answer to be 36 (33 + 3), but Flash sees that you defined the second value as a string, and as you just learned, a string plus a number gives a string as the result, so Flash would set myAge to be a string value of "333". Now imagine that the second value was a variable that you'd set to a string somewhere else in the movie, possibly by mistake, and you can see how easy it is to get confused.

> *Note that if you kept to the game plan and used the variable name* myAge_str *if you actually wanted a string value in the* age *variable, you would have smelled something fishy about the preceding line—the variable* myAge *doesn't look like it should be assigned a string value.*

Determining and defining the type of a variable

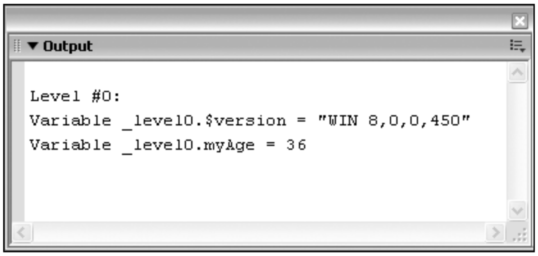
Wouldn't it be handy if you could just ask Flash what type a variable is? Well, luckily you can. You'll use the preceding example as your starting point.

1. Open a new Flash document and type the following lines into the Actions panel for the first frame:

```
var myAge;
myAge = 33 + 3;
```

The first line creates the variable myAge, with the var action, but doesn't give it a value. The second line is an expression that sets your newly created variable to some value.

2. Press Ctrl+Enter/Cmd+Return to test-run the movie. As you'd expect, there's nothing on the screen, but Flash has stored your variable away in its memory. To see this, go to the Debug menu and choose List Variables. You should see a box like this appear:



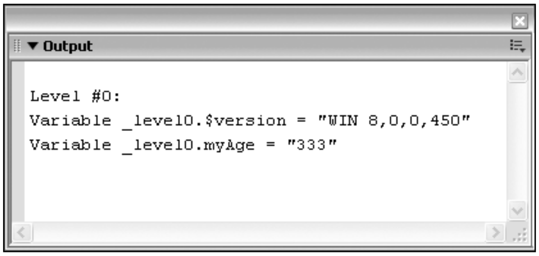
```
Level #0:
Variable _level0.$version = "WIN 8,0,0,450"
Variable _level0.myAge = 36
```

As you can see, the final entry contains your variable (don't worry about the first two entries or the `_level0` bit for now). You can clearly see that your variable is set to be the number 36.

3. Now go back to the Actions panel and change the line so that it reads as follows (forgetting for a moment that you should really be using the variable name `myAge_str`):

```
var myAge;
myAge = 33 + "3";
```

Now when you test your movie and look at the variables, you should see this:

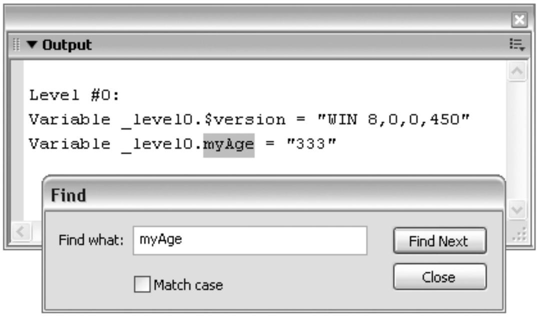


```
Level #0:
Variable _level0.$version = "WIN 8,0,0,450"
Variable _level0.myAge = "333"
```

You can see that your variable is now a string (notice the double quotes) and that it's set to the value `"333"`.

Well, that's one way of checking. Once you start building more complicated movies, though, it will become much harder to find the variable you're looking for in this list. That doesn't make this technique useless, though—in fact, should you ever need to do this, you'll find a handy Find command hidden away in the Output window's menu (located in the top right of the panel), which allows you to search the entire contents of the Output window for whatever it is you're looking for.

4. To test this, select Find from the Output window's menu and type in myAge. Click Find Next, and the variable will be highlighted for you.



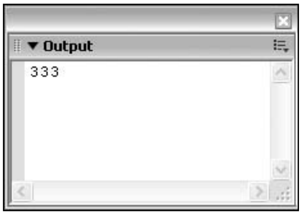
```
Level #0:
Variable _level0.$version = "WIN 8,0,0,450"
Variable _level0.myAge = "333"
```

> *Note that the* Find *window allows you to match case. Clearing the* Match case *check box is a good way to find any variable names that have been entered incorrectly—for example,* myage *instead of* myAge.

5. Close the SWF window and go back to the Actions panel. Add the following line just after the first two:

```
var myAge;
myAge = 33 + "3";
trace(myAge);
```

The trace command is one of the most useful commands in Flash to help you debug your ActionScript. If you run the movie now, you'll see that the Output window appears with just the contents of your variable in it:



```
333
```

This is helpful enough as it is, but it's not all you can do. To the casual observer, that still looks like just a number. Let's see if you can be a bit more explicit.

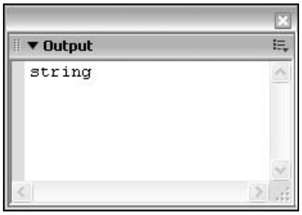
6. Go back to the Actions panel and alter your third line as follows:

```
var myAge;
myAge = 33 + "3";
trace(typeof (myAge));
```

Be careful with the parentheses here—if you put in too many (or too few), Flash will generate an error.

The typeof command tells Flash that you want it to give you the type of the variable that you've supplied as an argument. The whole of this expression is still taking place within the trace command, so it will still output whatever the result is.

If you test the movie now, you should get the following:



```
string
```

Now you can definitely see what type the variable is!

7. Try changing the variable back to how it was originally:

```
var myAge;
myAge = 33 + 3;
trace(typeof (myAge));
```

Then test your movie and see if the Output window contains the expected results.

So, you can now find out what type a variable is when things go wrong, but wouldn't it be better if there was a way you could prevent things from going wrong in the first place? Well, there is—if you tell Flash the type of your variable when you define it.

8. Change the first and last line of the code as follows:

```
var myAge:Number;
myAge = 33 + 3;
trace(myAge);
```

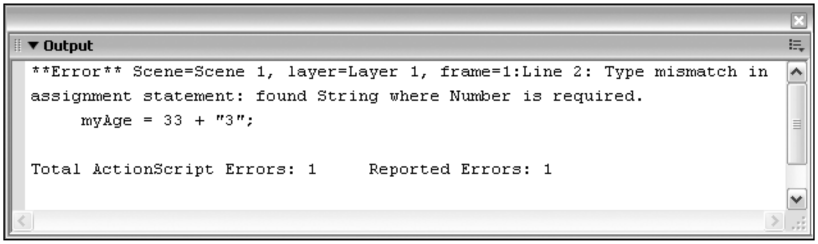
Do you remember us telling you how choosy Flash is about the case of variable names? Well, be aware that the variable type, such as Number or String, is *always* capitalized.

This time, you define the type of myAge as well as its name when you define it—you want it to contain numbers only. So what happens if you try to get Flash to store a string in myAge? Only one way to find out . . .

9. Change the second line of your code as shown:

```
var myAge:Number;
myAge = 33+"3";
trace(myAge);
```

When you run this code, Flash will realize that you've tried to store something in myAge that it wasn't defined to hold, and it will raise an error. Although the code here is less flexible—you can't store the string value "thirty-five" in myAge, for example—it does stop you from creating code that would place a string value in myAge by mistake.



```
**Error** Scene=Scene 1, layer=Layer 1, frame=1:Line 2: Type mismatch in
assignment statement: found String where Number is required.
     myAge = 33 + "3";

Total ActionScript Errors: 1     Reported Errors: 1
```

Next up, you'll investigate more ways in which you can use your newfound variable friends—more specifically, by using them to increase user interaction.

Input and output

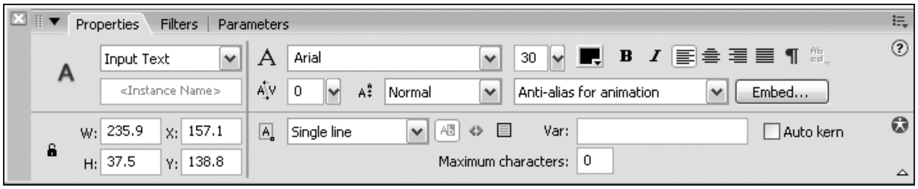
So far, we've talked only loosely about what variables are and the kinds of values they can store. It's about time you had a proper example of using them, so that you can start to see what they can do for your movies and websites. You'll begin by using them to store strings, since the practical benefits here are most obvious.

The act of storing strings is closely associated with **input to** and **output from** Flash, simply because users generally prefer to interact with a computer using text rather than raw numbers. Most complex interactions with computers are done using text of some kind, from filling in an online order form to entering keywords into a search engine. Both tasks are much more involved than, say, entering numbers into a calculator, and this is where string variables shine.

Creating an input field

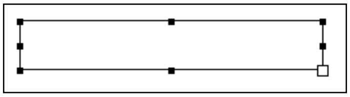
If you're going to use a variable to **store** a string that the user has entered, you need to enable the user to input a string.

1. Open a new movie and select the Text tool. Now turn your gaze to the Property inspector and make sure that Input Text is selected in the drop-down menu on the far-left side:



Input Text
<Instance Name>
Arial
30
0
Normal
Anti-alias for animation
W: 235.9 X: 157.1
H: 37.5 Y: 138.8
Single line
Var:
□ Auto kern
Maximum characters: 0

2. Click the stage to create a text field. If you've only worked with static text before, then you'll notice that two things are different.



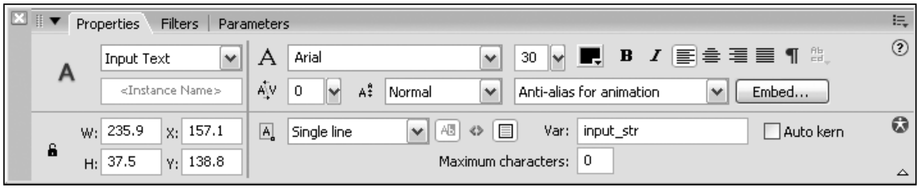
   First, a static text box has a circle at the top-right corner, whereas the **input text field** has a square at the bottom right.

   Second, a static text box will always start its life one character long and grow as you add more text to it. An input text field, on the other hand, immediately comes up with room enough to enter several characters. There won't be any text placed into it until after you've published the movie, so until then Flash needs to make a rough guess as to how much text you might need to enter.

   The default size is Flash's best guess; you can make the field longer (or even make room for more than one line of text) by dragging the little square on the bottom right of the text field.

3. Look at the Property inspector again. Now that you've assigned this box as an input field, Flash knows that you'll probably want to store that input in a variable for later use. To make this easy for you, it's provided a text box (labeled Var) where you can give it a name for that variable. Since this is your first input box, let's call it `input_str`.

   Also, while you're at it, click the Show Border Around Text icon (the icon immediately to the left of Var). Flash will draw a box around the text area's limits. Since this area will be used for inputting text, users will want to click it and make sure it's selected before they type anything in. The border will show users where to click.

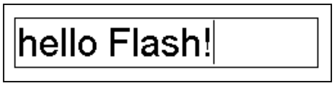


Input Text
<Instance Name>
Arial
30
0
Normal
Anti-alias for animation
W: 235.9 X: 157.1
H: 37.5 Y: 138.8
Single line
Var: input_str
□ Auto kern
Maximum characters: 0

> *The way you're using text fields at the moment isn't the normal or best way of using text fields in Flash 8. It is, however, the best way to show how variables work. You'll learn the proper way to use text fields (using event-driven code and the text property of text fields) when the time comes.*

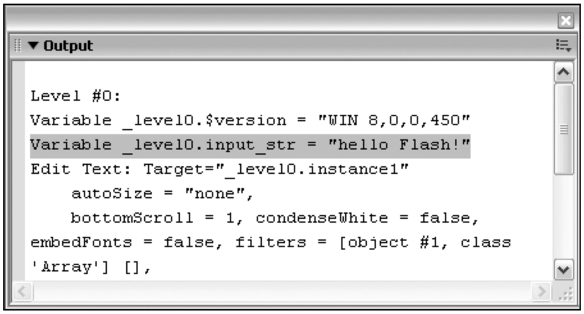
4. Now make sure that the text color isn't set to white. This may sound like a really obvious thing to do, but it's easy enough to forget and annoying if you run the movie and can't think of why you don't see any text appear in the text box.

5. Test the movie. You should see an empty rectangle on the stage. Click inside it, and a text-entry cursor will appear. Type in whatever you want. If you type in more than the box can fit, it will scroll to accommodate your input as you type. You can use Backspace, normal copy and paste keyboard combinations, and most of the other keys you'd expect, though you can't use Enter/Return to finish your input:



hello Flash!

Now it may appear that Flash isn't doing anything when you type, but it's actually storing whatever goes in your text field inside your input_str variable. At this point, you could look up input_str in the variables list and find (sure enough) that input_str contains the string literal "hello Flash!":



```
Level #0:
Variable _level0.$version = "WIN 8,0,0,450"
Variable _level0.input_str = "hello Flash!"
Edit Text: Target="_level0.instance1"
    autoSize = "none",
    bottomScroll = 1, condenseWhite = false,
embedFonts = false, filters = [object #1, class
'Array'] [],
```

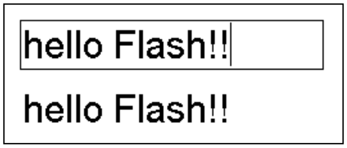
Creating an output field

1. Duplicate the existing text field and move the new copy below the original. With the new copy selected, bring up the Property inspector and change the Text Type drop-down menu (located top left) to Dynamic Text. This time, make sure the Show Border Around Text and Selectable icons are **not** selected:

   You deselect these icons because the field that you're about to create is for use by Flash, not the user. It's being used as a display area, so the user won't need to see it drawn or be able to select it.

2. You're going to leave the variable name for the duplicate text field as input_str for a good reason. Whereas for the input field the variable name tells Flash to take whatever is in the text field and put it into your variable, for the dynamic text field the variable name does the opposite: it takes whatever is in the variable and displays it in the text field.

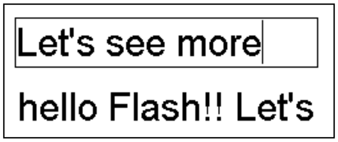
3. Now test the movie. You'll see the same empty rectangle, until you start entering some text. When you do that, the dynamic text field will begin to "echo" what you've entered:



hello Flash!!

hello Flash!!

You'll also see, though, that it's of fixed length, and it won't scroll in the same way as the input box to display everything that you type in:



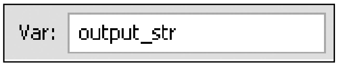
Let's see more

hello Flash!! Let's

The update speed of both text fields is determined by the frame rate specified for the movie. At the default frame rate of 12 frames per second (fps), the speed of text entry and its echo should look immediate. If you go and change the frame rate via Modify ➤ Document *to 1 fps, you'll see a marked change in the update rate when you test the movie again. When you're done experimenting, return the frame rate to 12 fps.*

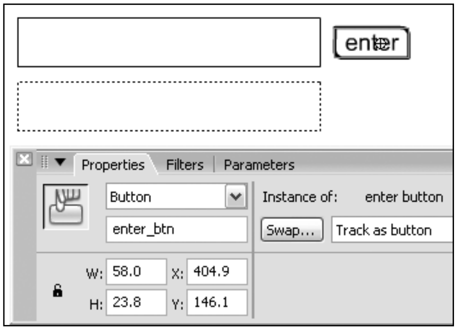
At the moment, Flash is displaying a copy of what you type into the input box in real time, as soon as you type it in. If you make a mistake, Flash displays it for the world to see. You need a way to make Flash wait until you have finished entering your text and are happy with it before considering it the finished article. The easiest way to give you this extra feature and expand on what you've got is to create an enter button.

4. Select the lower text field (the output box), and change its variable to `output_str`:



Var: output_str

5. Next, add a button next to the input text field and use the Property inspector to give it the instance name `enter_btn`. You should know how to create your own buttons by now, so we'll leave the details to you (or you can use the download file `inputOutput.fla` if you need a bit of revision). However you do it, make sure that when you specify the enter text for your button, the text is set as static (in other words, the Text Type drop-down is not still set to Dynamic Text in the Property inspector):

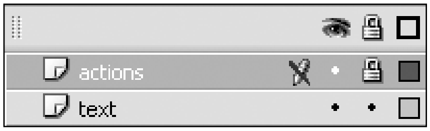


Button
enter_btn
Instance of: enter button
Track as button
W: 58.0 X: 404.9
H: 23.8 Y: 146.1

You now need to wire up the button with a little ActionScript. You'll do this by defining an event handler for the button, just like you did in Chapter 1.

6. Name the existing layer text and add a new layer called actions above it.

7. Lock the actions layer and select the first frame in this layer.



8. Select the first frame in the timeline and bring up the Actions panel. You'll start off by putting in the framework for an `onRelease` event on the `enter_btn` button:

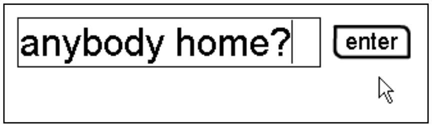
```
enter_btn.onRelease = function() {
};
```

Remember that a benefit of locking a layer is that doing so locks only content related to the layer that's on the stage. You can still add scripts to a locked layer, and you can still edit the timeline (add or remove frames and keyframes). The only thing you can't do is add content to the stage on the locked layer. The actions *layer is usually the layer you want to reserve for code only, and locking it permanently does just that.*

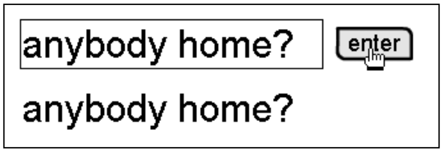
9. The next step is to write some event handling code, so that something actually happens when the user releases the button and triggers the onRelease event. Add the following line:

```
enter_btn.onRelease = function() {
  output_str = input_str;
};
```

10. Now test the movie. This time when you enter any text, the bottom field doesn't echo your input . . .


anybody home?

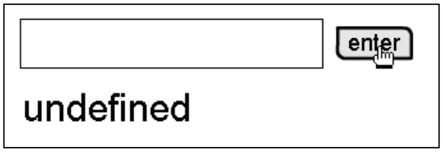
. . . until you click the enter button:


anybody home?

anybody home?

11. Make sure you hang on to this FLA because you'll use it several times later as a simple Flash user interface. Save it with the filename inputOutput.fla.

12. Test the SWF again, but this time click the enter button *before* you type anything into the input text field.


undefined

You haven't defined anything about your two variables input_str and output_str, so when you click the enter button, you get undefined back because that's exactly what's currently in the input field. You can prove that undefined and "nothing" aren't the same thing by running the SWF again, and entering a character and then pressing the Backspace key. Although there's nothing in the input text field, this time you have *defined* the contents of the text field (and therefore, input_str). More subtly, even though there's nothing in the text field, you've actually defined what **type** of nothing that nothing is—in this case, it's "" (a string of 0 length). This time if you click the enter button, you'll get back the contents of the top text field; rather than undefined, you'll see a blank.

If you think about it, such a case would very quickly become a problem if you wrote any code that read the value of the input text field, such as for a website that sells cars. If your input text field was the bit where a buyer added the required car color, the buyer could end up ordering a car with color undefined. Better than mauve, we guess, but probably not what was intended—in fact, in most cases the unexpected appearance of undefined would be a bug in your code.

13. To fix this, you have to explicitly define all the variables in your code. Add the following new lines and test the SWF:

```
enter_btn.onRelease = function() {
  output_str = input_str;
};
var input_str:String;
var output_str:String;
```

Hmm. Looks like a step backward, not forward! This time *both* text fields show undefined, and they do it before you click anything. Despite appearances, this is actually a *very* good thing because you know something is wrong immediately, as opposed to the last version of the FLA, which sat there hiding the bug until you performed some specific steps.

> *The new listing uses well-defined variables, and that's why the error appears quickly. This is a good thing because it tends to create workflows that result in better quality code, which is what we'll always aim for in this book's techniques, as opposed to showing quicker ways through.*

So what's the problem? Well, last time around (i.e., using the FLA without the var lines) you didn't create input_str and output_str until the first time the line output_str = input_str; was run. This time around, you define the two variables straight away using var, and this reveals the problem immediately. The only remaining issue is that you've defined the variables, but not their contents.

14. Add the following lines:

```
enter_btn.onRelease = function() {
  output_str = input_str;
};
var input_str:String;
var output_str:String;
input_str = "";
output_str = "";
```

This time, you create the variables and also define their contents. There's no chance of undefined appearing (although you can get it to show up "undefined" if you type it in, which is the text string undefined, not undefined).

You'll create a lot of variables in this book, so it would help if you could define them a bit quicker than this. The following code lines use var to both create variables and give them initial values:

```
enter_btn.onRelease = function() {
  output_str = input_str;
};
var input_str:String = "";
var output_str:String = "";
```

As you've learned here, creating variables is only part of being ready to use variables—you should also give them definite and well-defined values. If you don't, you'll probably come away with shorter code, but that code will also be more likely to have bugs hiding in it.

The process of creating variables and giving them starting values (also termed **initial values** by programmers) is called **initialization**. When you use Flash's strict typing (as you will throughout this book), it's very important that you always properly define your variables by doing the following:

- Give them a name (e.g., `input_str`).
- Give them a type (e.g., `String`).
- Give them an initial value, even if this is just `""` or `0` (a number zero).

If you're working with your own files, save this FLA as `inputOutput2.fla`. You'll use this file as the starting point of another example later on in the chapter.

Using string expressions

Let's recap. To start putting together the first building blocks for inputting and outputting text within Flash, you've created simple text fields for obtaining and displaying the values of basic variables. Now that you know how to do this, your available lines of communication with the user (and levels of interactivity) have gone from basic mouse clicks to phrases and sentences.

Now that you have your text input and output working, in this section you're going to look at how you can use string expressions to give Flash a more human face and communicate with users on a personal level.

Working with strings

In this exercise, you'll create a string expression that combines the value of the variable `input_str` with some extra text to display more than users might expect.

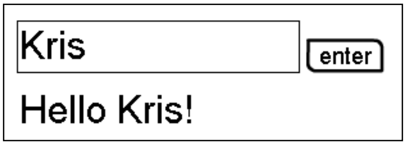
1. Open the FLA that you've just been working on (or use the file `inputOutput2.fla` from the download file for this chapter).
2. Select the first frame of the actions layer (if it isn't already selected) and open the Actions panel. Then select the line `output_str = input_str;` and alter it so that it matches this:

```
output_str = "Hello " + input_str + "!";
```

   Make sure that you include the space after `Hello`, otherwise Flash will just crunch all the words together.

   When ActionScript encounters this, it takes the string literal `"Hello "`, adds the contents of the variable `input_str`, appends the second string literal `"!"`, and stores the result in the variable `output_str`. This may sound confusing, but rest assured it will all make sense when you see it in action!

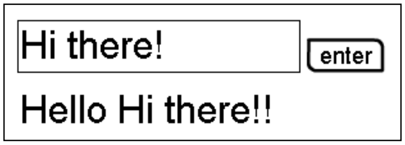
3. Test the movie and enter your name. When you click the enter button, Flash should greet you like a long lost friend:


Kris

enter

Hello Kris!

While we don't want to undersell what you've done here, we have to point out that there's a slight problem: you haven't been able to empower Flash with the grammatical knowledge to check whether what it's saying to you makes sense. Whatever you type in, Flash will give it back to you as part of the same answer:


Hi there!

enter

Hello Hi there!!

We can't say that we have the solution to this for you here. We've shown you how to use string expressions to build in a degree of personalization, but be aware how quickly the illusion of intelligence falls apart. If you try this within a website project, be careful. Still, it's reassuring to know that computers still can't be more intelligent than we are!

Virtually all the variable examples you've looked at so far have dealt with storing strings. Of course, that's not the only type of data you can put into an ActionScript variable, so let's take a closer look at the other two types: **numbers** and **Booleans**.

Working with numbers

In everyday use, numbers can represent more than just raw values. Numbered seats in a theater don't represent a chair's **value**, but its **position**. The ISBN for this book doesn't represent its value or its position, but is a **reference** or **index** to the book.

To a much greater extent than strings, numbers lend themselves to being operated upon: they can be added to, subtracted from, multiplied, divided, and so on. If you store numbers in variables, all kinds of possibilities open up.

Returning to those theater seats, if you arrange to store the number booked so far in a variable called seatsTaken, then the number of the next seat available (assume you sell them in order) for booking is *always* going to be seatsTaken+1, regardless of the actual value stored in the variable. Look at this:

```
seatsTaken = seatsTaken +1;
```

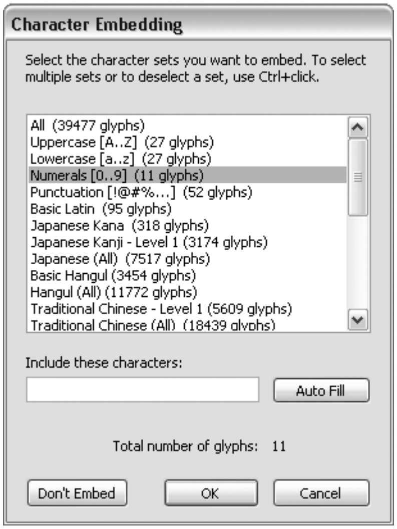
This line says "Add 1 to the current value of seatsTaken and store the result back in seatsTaken." Each time ActionScript encounters it, the number stored in the variable increases by 1. Remember, variables are all about giving ActionScript memory.

Let's try some of this out, using the interface FLA again. You can use inputOutput3.fla from the download file if you forgot to save it. In this exercise, you're going to get the user to input some numbers, and then Flash will perform some calculations on them.

Performing simple calculations

In the previous exercise, the values you saw going into input_str were actually string values, which is to say that they could include more than the numbers 0 to 9 and the decimal point that you would normally expect in numeric values. Before you start using input_str in numeric expressions, you need to take steps to prevent the user from entering characters that could spoil your work.

1. Select the input text box and turn your attention back to the Property inspector. Click the Embed button next to the Var field to bring up the Character Embedding panel (as seen to the right). Then highlight Numerals [0..9] (11 glyphs) and click OK. This allows you to enter the numbers 0 to 9 plus the decimal point.
2. Now test the movie. You'll find that you're no longer able to enter anything other than decimal numbers, which is a step in the right direction.



Character Embedding

Select the character sets you want to embed. To select multiple sets or to deselect a set, use Ctrl+click.

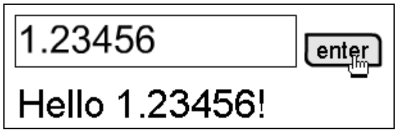
All (39477 glyphs)
Uppercase [A..Z] (27 glyphs)
Lowercase [a..z] (27 glyphs)
Numerals [0..9] (11 glyphs)
Punctuation [!@#%...] (52 glyphs)
Basic Latin (95 glyphs)
Japanese Kana (318 glyphs)
Japanese Kanji - Level 1 (3174 glyphs)
Japanese (All) (7517 glyphs)
Basic Hangul (3454 glyphs)
Hangul (All) (11772 glyphs)
Traditional Chinese - Level 1 (5609 glyphs)
Traditional Chinese (All) (18439 glyphs)

Include these characters:

Auto Fill

Total number of glyphs: 11

Don't Embed | OK | Cancel



1.23456 | enter

Hello 1.23456!

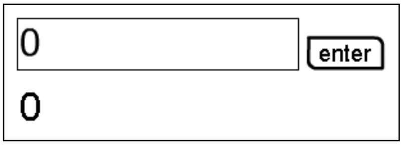
3. Close the SWF and select frame 1 on the actions timeline. Call up the Actions panel. Now that you know it's safe to treat input1 as a numeric value, you can perform a bit of simple arithmetic on it and assign the result to output1. Change the code to read as follows (changes highlighted as usual):

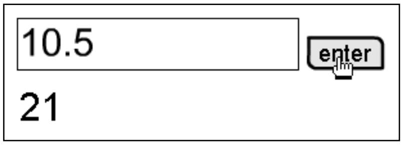
```
enter_btn.onRelease = function() {
  outputNum = inputNum * 2;
};
var inputNum:Number = 0;
var outputNum:Number = 0;
```

> *Computer programming languages use the asterisk (*) to represent multiplication in place of the traditional ×. The idea is to remove the chance of confusion with the letter "x".*

4. You also need to change the variable names associated with the two text fields. Select each in turn and change input_str to inputNum, and output_str to outputNum.

5. If you now test the SWF, you'll start with zeros in the two fields. If you change the top zero to a number, the lower one will change to twice the entered value:

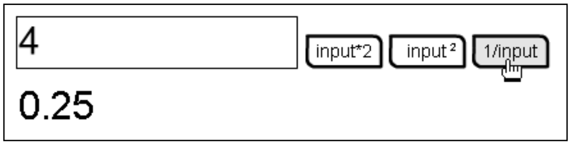




If you like, you can try changing the expression (or even adding buttons with new events of their own) to give the following information:

- `outputNum = inputNum^2;` for the square of the amount entered
- `outputNum = 1/inputNum;` for the reciprocal of the amount entered

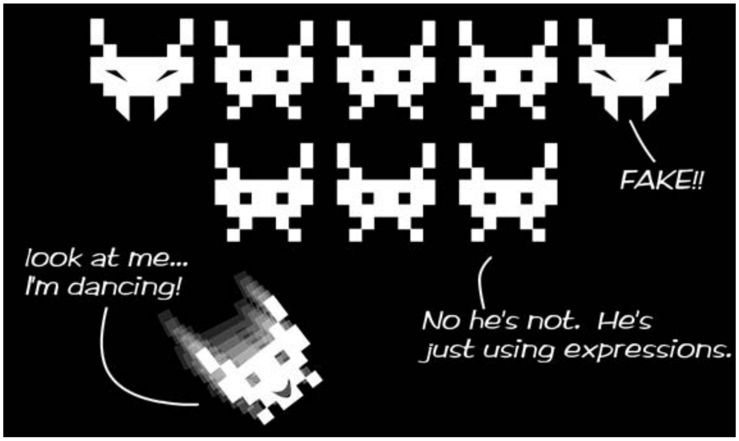
Here's a shot from one you can find in the download file saved as `calculator.fla`:



Other uses for numeric expressions

Numeric expressions have a lot more to offer than just making Flash do sums for you. For example, you could use the results of numeric expressions to control the position or orientation of an instance, allowing you to create some very complex movements. You could have an expression to tell your alien spaceship, Blarg, where to move after it makes its bombing run on a game player.

You can also use numeric expressions to jump to frames within a movie based on **dynamic** values, and not just to the fixed number values you've used so far. This allows you to control timeline flow to a much greater degree. For example, you could jump to an explosion if the player is hit by an alien.



Some of the stuff you've looked at so far may have seemed a little removed from space invaders flitting about the screen and cool, ActionScript-heavy dynamic sites, but you're learning about the cogs that make these wheels turn. You'll continue to do that in the next section.

Working with Boolean values

If you buy a packet of cookies, you might do so because they look tasty and you're not on a diet. This is a very clear-cut decision: if two criteria are met, then you buy the cookies. In general, however, we don't tend to make such straightforward decisions; instead, we base our decisions at least partly on a comparison, our mood, or a predisposition. We might even base some decisions on inputs that have nothing to do with the choice at all—for example, "I don't want to talk to him because I don't like the color of his shirt." Our decisions can have many more outcomes than simply "I will" or "I will not."

Computers, though, stick to decisions of the more clear-cut variety, which is a relief because a computer probably tracks your bank balance and transactions. You wouldn't like your balance to halve suddenly because the computer was having a bad day and you turned up at the bank wearing a lilac shirt. Computers base their decisions on statements of the yes/no or true/false variety. Their decisions have only two possible outcomes at a time, although they can apply lots of yes/no decisions in a row to tackle more complex outcomes.

Suppose, for example, that you wanted to find out whether a number was higher or lower than 100. It could be the price of a share on the stock market. If the number is lower, you'll think about buying, and if it's higher, you'll think about selling. Depending on which side of 100 the price is, you'll perform one of two different sets of actions. Alternatively, it could be that you're writing a space invaders game in which the invaders move down the screen in an attempt to land on the player's planet. Once the distance of any invader from the top edge of the stage is greater than 100, you know it's at the bottom of the screen, and the player has lost.

Take the first case and assume that the share price you're interested in is stored in a variable called `price`. Now have a look at this:

```
buy = price < 100;
```

When you've seen code like this in the past, the expression on the right side of the equal sign has been evaluated, with the result being stored in the variable on the left side. But what *is* the value on the right side? Given the title of this section, you won't be surprised to discover that it's a **Boolean value**, which means that `buy` will be set to `true` if `price` is less than 100 or `false` if `price` has any other value. Flash's thinking goes like this:

- If the price is less than 100, then set `buy` to equal `true`.
- If the price isn't less than 100 (i.e., it's greater than or equal to 100), then set `buy` to equal `false`.

`true` and `false` are the **only** Boolean values, and they're the only possible values on the right side.

> *Note that* `true` *and* `false` *here are neither strings nor variables. They're simply the two possible Boolean values. If you accidentally place quotes around either a* `true` *or* `false` *value (rendering it as* `'true'` *or* `'false'`*), they will actually become string variables, and therefore won't be Booleans at all!*

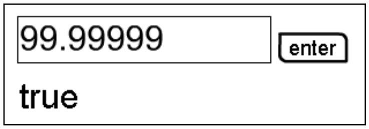
Testing whether statements are true

Until you start to look at conditional operators in the next chapter—which will indeed allow you to perform different actions depending on Boolean values—there's a limit to what you can do with them. However, you can at least check that what we've been saying is true and use them in some slightly more complex expressions. Let's modify your movie again.

1. Load inputOutput.fla (the file you saved earlier), making sure it's free of all the little tweaks you subsequently made to it. Alternatively, you can open greetings.fla. Select the input text and change the selection in the Character Embedding panel again, so that you're only accepting numbers and a decimal point.
2. Now select the first frame in the actions layer and call up the Actions panel. Change the code it contains as follows:

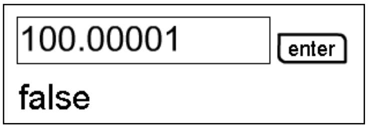
```
enter_btn.onRelease = function() {
  outputBool = (inputNum < 100);
};
var inputNum:Number = 0;
var outputBool:Boolean = true;
```

3. You also need to change the variable names in the text fields, so select each text field in turn, changing the Var field in the top text field to inputNum and the lower one to outputBool.
4. Run the movie, and you'll see that Flash now looks at any number you enter and compares it with 100. If it's lower, inputNum<100 is a true statement, so you get true:


99.99999

true

If it's greater than or equal to 100, inputNum < 100 isn't true, so you get false:


100.00001

false

Logic operators

Think back to high school when you had lessons on electrical circuits. You may remember that there were logic gates that would open only if their conditions were made. These gates had crazy names like AND, OR, NOT, and NAND, and similar **logic operators** exist in ActionScript. Here's a list of the ones that it recognizes:

- &&: AND
- ||: OR
- !: NOT

If you need to, you can use these logic operators to make your test more selective. That's what you're about to do.

Take the earlier example about share prices. If you were prepared to buy shares if the price per share was below 100, then you might want to stop buying if the price went below 80, because such a sharp drop would indicate unfavorable market conditions. In plain text, you would need something like this:

If the price is less than 100 AND greater than 80 . . .

Bringing this closer to the expression in this example, you need this:

```
(inputNum < 100) AND (inputNum > 80)
```

Flash uses the && symbol to signify AND, so the ActionScript you actually want is this:

```
(inputNum < 100) && (inputNum > 80)
```

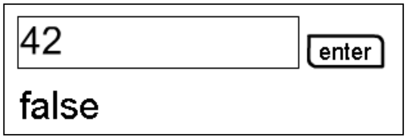
Testing for more than one thing at a time

Let's change the script to test for the double expression that you looked at earlier.

1. Alter your frame 1 script to read as follows:

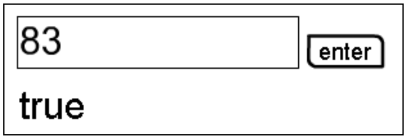
```
enter_btn.onRelease = function() {
  outputBool = (inputNum < 100) && (inputNum > 80);
};
var inputNum:Number = 0;
var outputBool:Boolean = false;
```

2. Now run the movie. You'll see that you get a true result only if the price is between 80 and 100.



42

false



83

true

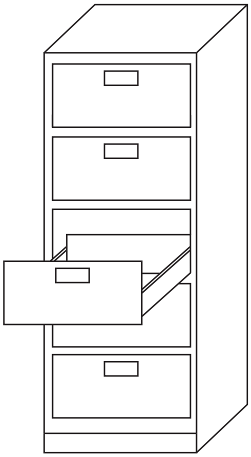
Using similar techniques, you can build up very complex conditions that involve a number of other operators and terms, and you'll start using them for real in the next chapter.

Before you do that, though, we're going to demonstrate how you can get more out of your variables by using arrays.

Arrays

The easiest way to think of an **array** is as a container for a collection of variables, which you access using a single name. From the outside, it looks just like a single entity, but it can hold lots of different values at any one time.

In many ways, an array isn't dissimilar to a filing cabinet: the cabinet itself has a name, but it may contain several drawers, each of which can contain its own information. Each drawer is like a variable in its own right, but it's stored in a cabinet along with other drawers, because they all hold data that's related in some way.



When you're looking at a variable array, you need to specify two things:

- The **name** of the array
- The **index** (or offset number) of the **element** (or "drawer") you want to use

Say you have an array called `myColor` that contains five pieces of information—that is, five elements. To refer to the first one (that's the bottom drawer in the cabinet), you use this:

```
myColor[0]
```

Note that the index (or offset) is surrounded by square brackets `[ ]`, and yes, the first array location is 0, not 1 as you might have expected. If the fact that there is a 0th drawer confuses you, think of it this way: whenever you see an index value of `n`, you're referring to the drawer that's offset by `n` from the bottom. In theory, there's nothing stopping you from leaving it empty, but this isn't recommended, for practical reasons you'll learn about later on. It may seem a little odd to begin with, but you'll soon get used to it!

So, an index of 0 gives you the base element. The second element (the first drawer up from the base) would be

```
myColor[1]
```

The third element is

```
myColor[2]
```

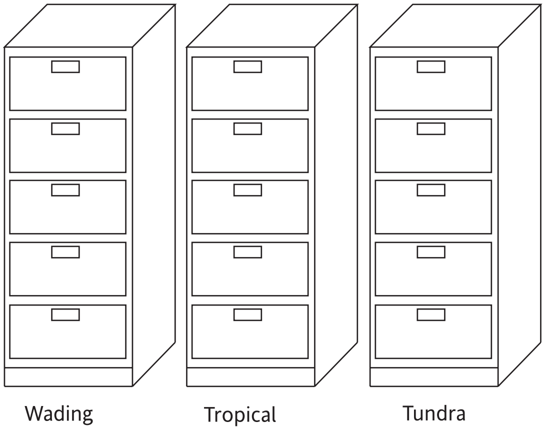
and so on. Likewise, say you want to get some information that you know is stored in the third drawer from the bottom of a filing cabinet called "contracts." In ActionScript terms, that filing cabinet is an array called `contracts`, and the third drawer can be accessed as the array element `contracts[2]`.

Reasons for using arrays

There are several reasons why it can be useful to store variables in arrays, as described in the following sections.

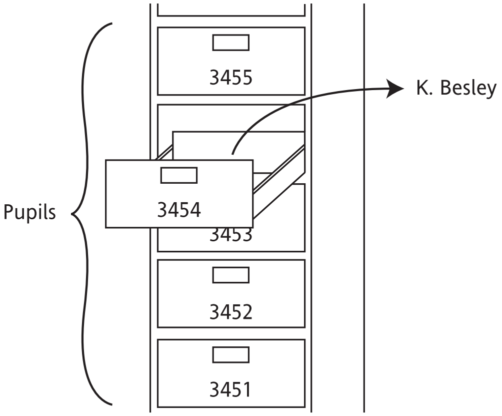
Arrays let you store related information together

Say you're working with a list of birds, and you want to keep together information on all birds that live around water. You might put them into an array called wadingBirds, and the first element might contain the string "flamingo". You could have other arrays such as tropicalBirds (containing "parrot", "hummingbird", and so on) and another called tundraBirds (containing "ptarmigan" and "arctic owl"). As you can see, borrowing that *Birds of the World* book from the school library at nine years old has left its mark!



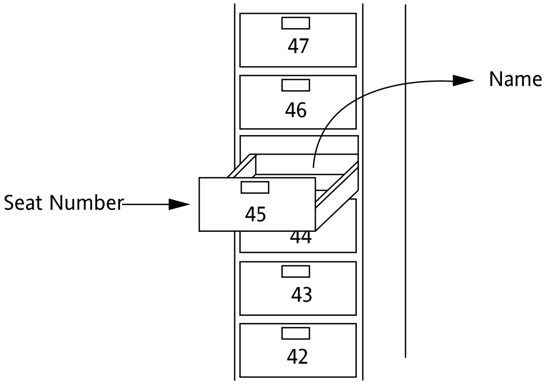
Arrays let you hold information in a specific order

Perhaps you want an array containing the names of pupils at a school, held in the order of exam rankings (best first, of course!). The variable pupils[3454] might well contain the name "K. Besley" . . .



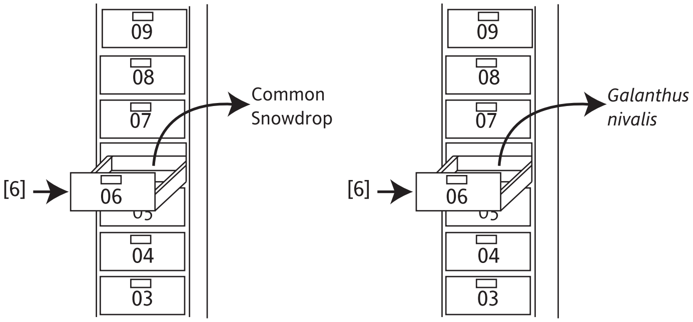
Arrays let you index information

For example, if you have an array to hold theater bookings, you could use the variable `seatingPlan[45]` to hold the name of the person who booked seat 45. In these circumstances, the numbering scheme is the most important thing, so you'd probably just ignore `seatingPlan[0]`—that is, unless the theater's seat numbering begins at 0 (a theater run by programmers, no doubt!).



Arrays let you link information

Say you have a list of common plant names and another list that gives their Latin counterparts, and you store them in a pair of arrays called `commonName` and `latinName`. Assuming the lists have been arranged properly in the first place, you could cross-reference between the two. For example, say you look in `commonName[6]` and find "common snowdrop." You can then look at the equivalent variable `latinName[6]` in the other array to find the Latin name *Galanthus nivalis*.



Creating a new array

Before you can do anything useful with arrays, you need to learn how to create them and pipe in information. Well, the first bit's simple enough. Say you want to create an array called `my_array`:

```
my_array = new Array();
```

The standard suffix for an array is (surprise, surprise) _array.

No problem! You now have an array, but it doesn't have any elements—it's like a filing cabinet without any drawers. It's easy enough to add them in, though:

```
my_array[0] = "Bill";
my_array[1] = "Pat";
my_array[2] = "Jon";
my_array[3] = "Tom";
```

Yes, you can assign values to the elements just as if they were plain old variables. Flash simply expands the array as you add new entries to it. You don't even need to use the same variable type for the different elements. Going forward from the previous example, you could easily write

```
my_array[4] = 1981;
my_array[5] = 1984;
my_array[6] = 1987;
```

This isn't the only way to define an array, though. You could write

```
myScores_array = new Array(1, 45, 31415);
```

or

```
myPets_array = new Array("cat", "dog","rat");
```

Note that if you set myArray[19], you're creating a *single* element. Flash *won't* create array entries [0] to [18] as well, so you have "missing drawers" that show up as undefined. This could be dangerous if subsequent code expects to see them filled with a sensible value (such as 0 or "").

It's much safer if you define each element right after you define the array. You would give each entry a default value. For example, the default value might be "" if you wanted an array of strings or 0 for an array of numbers.

In fact, there are various other ways to create arrays, but we recommend sticking to these for now—they should easily meet your demands.

Also worth noting is that the elements of an array can contain a mixture of the three different types of variables: number, string, and Boolean. The following is perfectly legal:

```
my_array = new Array();
my_array[0] = "Bill";
my_array[1] = 3;
my_array[41] = false;
```

Typing an array

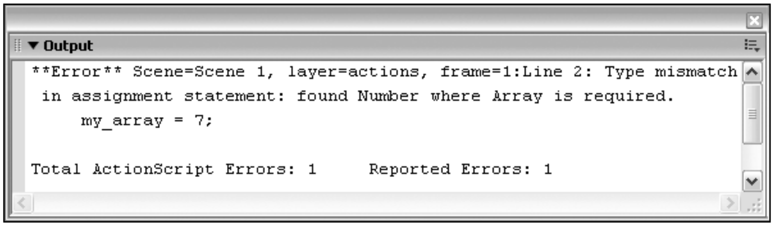
An array has its own type, Array, which contains elements that can be numbers, strings, or Booleans. If you want to type your arrays when you define them, you should use the following syntax:

```
var my_array:Array = new Array();
```

The advantage of this is that Flash won't let you destroy your array by equating it to something else (number, string, or Boolean). If you did this:

```
my_array = 7;
```

all of the other definition lines apart from the preceding typed one would overwrite your entire array structure with the value 7—not something that you want to happen often! The typed version would raise an error:



```
**Error** Scene=Scene 1, layer=actions, frame=1:Line 2: Type mismatch
 in assignment statement: found Number where Array is required.
     my_array = 7;

Total ActionScript Errors: 1     Reported Errors: 1
```

As usual then, typed array definitions are much better than untyped ones, because they force Flash to give you more possible errors when you compile your SWF, and this leads to more robust code.

Using variable values as offsets

One particularly useful feature of arrays is that you can specify offsets in terms of other variables. So, you can use an existing variable (defined completely separately from the array) to specify which drawer you want to look in.

Let's take a look at a quick example:

```
var myColor_array:Array = new Array();
var offsetNum:Number = 0;
myColor_array[offsetNum] = "green";
```

Here, offsetNum is a number variable set to 0, and you use it to specify which element in the array myColor_array should be given the value green. In this case, you've effectively said the following:

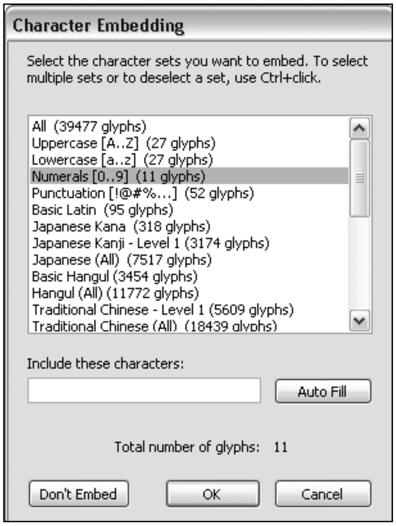
```
myColor_array[0] = "green";
```

This approach can be *very* powerful. By simply changing the value of your variable, you can use one bit of code to work on as many different drawers as your array has to offer. This means that you can write generalized code for manipulating array elements and specify the offset quite separately.

Using a number variable to select a text string from an array

Let’s use our faithful interface inputOutput.fla to take a quick look at arrays in action.

1. Open the FLA and select the top (input) text field. In the Property inspector, click the Embed button, and make sure the text field is configured to only accept numerals:



Character Embedding

Select the character sets you want to embed. To select multiple sets or to deselect a set, use Ctrl+click.

All (39477 glyphs)
Uppercase [A..Z] (27 glyphs)
Lowercase [a..z] (27 glyphs)
Numerals [0..9] (11 glyphs)
Punctuation [!@#%...] (52 glyphs)
Basic Latin (95 glyphs)
Japanese Kana (318 glyphs)
Japanese Kanji - Level 1 (3174 glyphs)
Japanese (All) (7517 glyphs)
Basic Hangul (3454 glyphs)
Hangul (All) (11772 glyphs)
Traditional Chinese - Level 1 (5609 glyphs)
Traditional Chinese (All) (18439 glyphs)

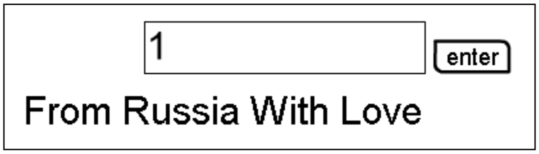
Include these characters:

Total number of glyphs: 11

2. With the top (input) text field still selected, change the contents of the Var field to inputNum.
3. Now select frame 1 of the actions layer on the timeline and make the following changes to the code:

```
enter_btn.onRelease = function() {
  output_str = bondMovies_array[inputNum];
};
var output_str:String = "";
var inputNum:Number = 0;
var bondMovies_array:Array = new Array ();
bondMovies_array[0] = "Doctor No";
bondMovies_array[1] = "From Russia With Love";
bondMovies_array[2] = "Goldfinger";
```

4. Make the lower text field longer if it looks like it won’t accept the length of “From Russia With Love”. Also right-align your lower text field.
5. Select the upper (input) text field and change its variable (the Property inspector’s Var field) to inputNum.
6. Now test the movie. When you enter a number in the first field and click the enter button, it’s used as the offset for bondMovies_array. As long as the number entered is between 0 and 2, you’ll be rewarded with the name of the relevant film:



1

From Russia With Love

Well, if you happen to count the films from 0, that is.

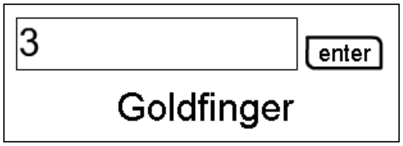
Let's face it, most people would think your system was broken if they fed in the number 1 and got *From Russia With Love*. *Everyone* knows that was the second movie! Don't they?

7. Let's fix this "broken" movie. Either you can change the array itself, leaving element 0 empty (or maybe hide *Casino Royale* away in there where nobody's likely to spot it) or you can tweak the offset. Let's do the latter for now:

```
enter_btn.onRelease = function() {
  output_str = bondMovies_array[inputNum-1];
};
```

By doing this, 1 is deleted from the entered value, so that a value of 1 becomes 0, 2 becomes 1, and so on. This way, the correct corresponding movie is retrieved from the array, so that an entered value of 1 retrieves *Dr. No* (the very first Bond movie) from element 0 of the array.

8. Now you can run the movie, type a number, click enter, and see the title you expected:



3

Goldfinger

Presto! Enter a number and get back a film title.

> *You'll see your old friend* `undefined` *if you enter 0 or anything greater than 3. Although that makes the code look unfinished, it's actually (again) a good thing to see* `undefined` *rather than a blank field or an incorrect value—it's a definite marker telling you this is something you have yet to address. Despite the code returning* `undefined`*, this doesn't represent a bug, but an expected value, and this is due in part to the fact that you're using* ***well-defined variables*** *throughout. Not only do you use meaningful* ***variable names****, complete with the appropriate naming style (camelCase and _suffixes), but also you define the* ***value*** *and* ***type*** *for each variable.*

Of course, we've barely touched on the power of arrays here, but hopefully you've seen enough to get a feel for what they are and what they do. You'll be seeing a lot more of them!

In the next couple of chapters, you're going to look at decision making and looping, topics that really make the most of what you can do with both variables and arrays. You'll look at the decision-making functionality that you can build using Boolean values, use ActionScript loops to act on whole arrays in one fell swoop, and even build a fully functioning hangman game to tie all these threads together!

Book project: Starting the Futuremedia site design

In the last chapter, you looked at how the Futuremedia site works in principle. Now it's time to begin creating the basic graphics for the site. Although you won't be adding any scripts in this chapter, you'll start using the drawing tools in a manner that you may not be used to yet but that is critical for ActionScript. You'll place everything on the stage precisely by entering numbers in the Property inspector. You don't need to do this when creating more general sites, but in a Flash-heavy site the ActionScript needs to know where everything is to pixel accuracy, so you have to be precise.

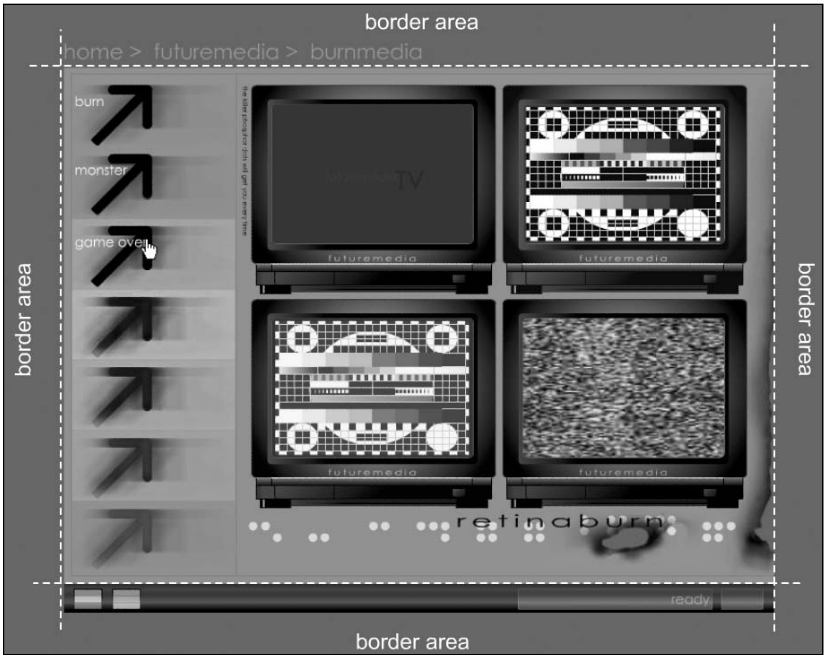
The final file created in this chapter is included as `futuremedia_setup1.fla` if you get stuck.

> *The Futuremedia site uses a font called Century Gothic. If you try to open any of the prebuilt FLA files on a computer without this font installed (this is pretty unlikely as it is a default on Mac and PC), you may get a warning message from Flash. If you see this, you should select* _sans *as the substitution font for now.*

What you're going to do

In this section, you'll

- Learn about and set up the Flash stage and timeline for your ActionScript site.
- Create the text areas that are in the border area of the site. This includes the text at the top-left corner of the site and the black bar at the bottom.



What you’re going to learn

The concepts to take on board for this section of the book project are as follows:

- The level of accuracy and placement ActionScript sites require of the graphics, which is far more than you may be used to when creating sites that use much less ActionScript and loads of tweening. The easiest way to manage this is by adding lots of construction lines onto the stage early on in your development, so you have a framework into which you can place your design.
- The amount of preliminary sketching you have to go through to get it all right first time. This involves time spent away from the main task every now and again, and summarizing what you want via a sketch is time well spent. If you find your design changing, be sure to create a new set of sketches to maintain an overall design-eye view of where you’re heading.

Choosing and setting the stage size

The Futuremedia site is designed for a screen size of 1280 × 1024 or more. This may seem a little high to some users, but bear the following in mind:

- We chose this size because it’s easiest on the eye during development on a typical 1280 × 1024 development machine.
- Pixel sizes are pretty much meaningless in Flash in any case. If you want the site to be viewable on a (for example) 800 × 600 screen, you’ll simply scale it during publishing.

> *As long as you create a stage size that’s in the approximate ratio 4:3 (the standard monitor ratio), you can scale it to target it for any screen size you want by changing the stage size when you publish (*File ➤ Publish Settings ➤ HTML *tab).*

For those unsure of the minimum viewable stage size that will work on all common browsers at the standard screen resolutions, here’s a quick table to help you (it takes into account both the Mac and PC):

Screen resolution	800 × 600	1024 × 768	1280 × 1024
Maximum Flash stage size	590 × 300	975 × 590	1230 × 845

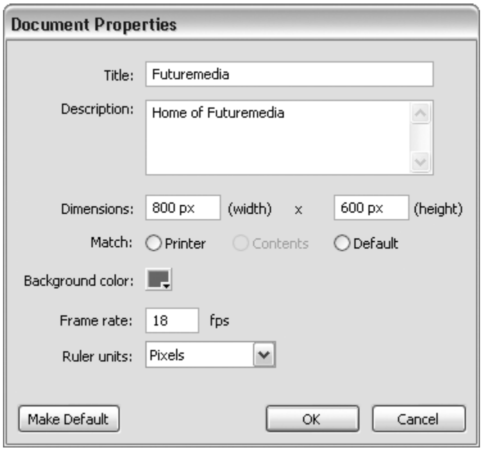
Unless you’re designing for portable devices, 800 × 600 is the minimum screen resolution you should assume. If a desktop can’t show this size screen, it probably can’t display Flash 8 content either!

Note that we haven’t included sizes for 1600 × 1280 or above. This is because such screen sizes aren’t widespread enough for you to safely design content for them.

You'll size your stage at 800 × 600, which is well within the 1230 × 845 stage limit for a 1280 × 1024 screen resolution.

Another, more subtle reason we chose 800 × 600 is related to performance issues. Flash really slows down if you create large sites that contain a lot of motion, and 800 × 600 is around the maximum stage size for a responsive Flash site. By setting your stage to this size during development, you're able to more easily identify performance bottlenecks and sluggish animation.

1. Open a new FLA with File ➤ New.
2. Select Modify ➤ Document to access the Document Properties window (you can also get to this window by double-clicking the current frame rate under the timeline).
3. Change the movie Dimensions to 800 × 600 and Frame rate to 18. Click the Background color brick and change it to the third gray selection down from the top, leftmost column (#666666).



Document Properties

Title: Futuremedia

Description: Home of Futuremedia

Dimensions: 800 px (width) x 600 px (height)

Match: ○ Printer ○ Contents ○ Default

Background color: ▨

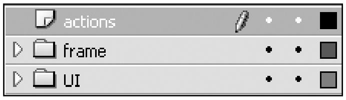
Frame rate: 18 fps

Ruler units: Pixels

Setting up the timeline

The next task is setting up the timeline for your proposed site. Although choosing layer names and setting up folders may seem trivial, remember that you're going to create a user interface that can handle a fairly large site and that will have lots of things whizzing around when it's up and running. A little thought and sensible planning at this stage works wonders!

1. Rename Layer 1 as actions and create two layer folders called frame and UI. Drag these folders below the actions layer (in case we didn't tell you before, it's always good practice to have your actions layer at the top of all the layers in your movie so you know just where to find it). The frame layer will contain all the symbols in the border area (or "the frame"), and the three strips (as discussed in the walk-through earlier) that form the main interface itself will be in the folder UI.

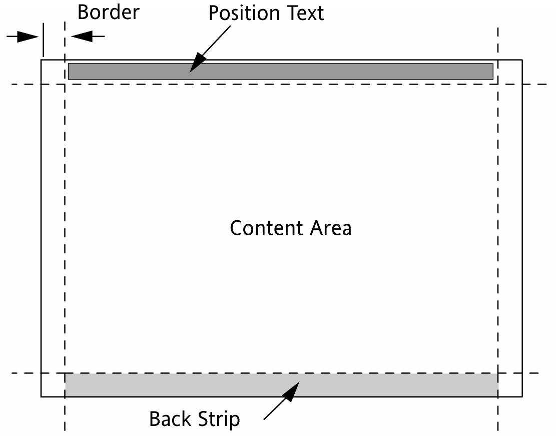


2. The actions layer is reserved for the code that will later animate your site. To make sure you don't inadvertently put graphics in this layer, lock the actions layer now.

> *On large site designs, a good tip is to change the layer color of the* actions *layer to black, to signify code (some people use white for this purpose). To change the layer color, double-click the color square and select the new layer color by changing the* Outline *color in the* Layer Properties *window that will appear.*

Creating layout guides

Although you already know what the site design will look like, here's a first sketch of the site design:

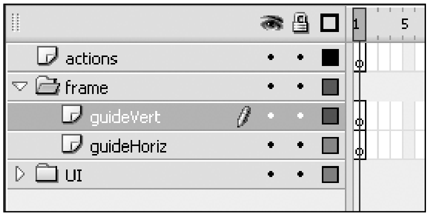


As you can see from this sketch, the dotted construction lines are fundamental to the layout—they separate the main content area (which will eventually contain the sneaky `tricolor/strip` movie clip combo) from the border areas containing the position text and back strip areas. Although it isn't shown in the diagram, the border width in the final design is 30 pixels. Why 30 pixels? Well, we wanted the design to be flexible enough for you to customize it as required, and 30 pixels is a good size for any icons or other interface items you may want to add in the left and right borders. It's also more than enough for the position text and back strip to remain legible if you decide the scale the site design down if you need to customize it to work well in a screen resolution of less than 1280 × 1024.

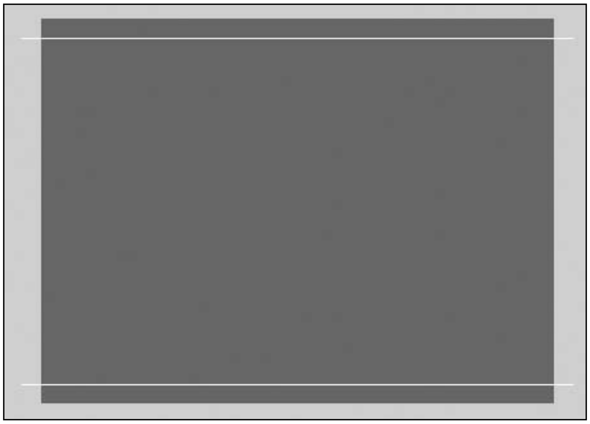
> *The sketch just presented (and others you'll see as you progress through the book) isn't a cleaned-up version of the initial sketch made on a piece of paper—it's the actual sketch created during the design process. The package of choice in creating such sketches isn't Auto Sketch or some other dedicated technical drawing application, but Flash itself. Using Flash, you can quickly mock up neat diagrams and sketches via a pen tablet, and you can allow Flash to straighten up your lines and curves as you go, so that you end up with a series of design documents that are fit to go straight into a book or show your client. In fact, many of the line diagrams for this book were created in Flash!*

It would be a good idea if you could transpose these construction lines onto your actual stage, which is exactly what you'll do next.

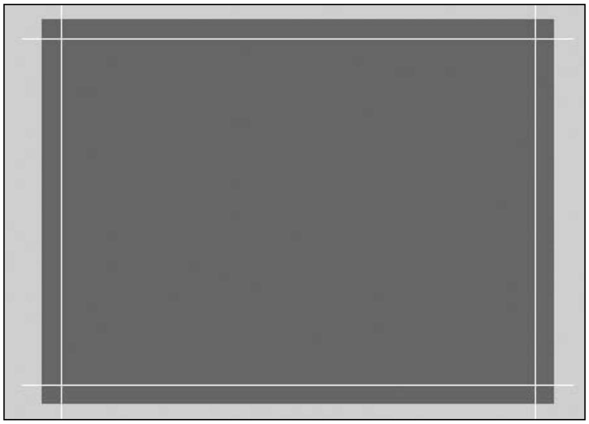
1. Create two new layers in the layer folder frame and call them guideHoriz and guideVert. Hide the grid (View ➤ Grid ➤ Show Grid) if it's displayed.



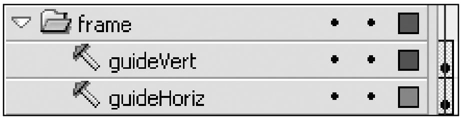
2. On the guideHoriz layer, draw out two thin horizontal lines that span the width of the stage, plus a little more. Now, using the Property inspector, give one line a y value of 30 and the other one a value of 570. Lock this layer so you don't mess up the lines during the next step.



3. On the guideVert layer, draw out two vertical lines in the same way as you did in the previous step, this time giving them x values of 30.0 and 770.0.



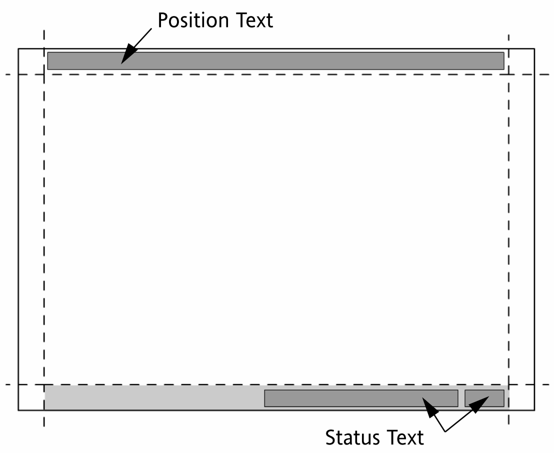
4. Make both of these layers guide layers (right-click/Ctrl-click the layer title and check Guide on the pop-up context menu that appears). You now have your border clearly marked on your screen. Lock guideVert so that both of your guide layers are now locked.



Note that the reason you aren't using ruler guides is that you can't place them accurately with the Property inspector. You want a greater accuracy than just doing it by eye. It's not that we're being overly fussy here—we know how particular ActionScript can be!

Adding the position and status text

At the top of your site is the position text, and this tells users where they are in the navigation (you'll get a better idea of what this does by looking at the final site design). Here's the second original design sketch for this site, which now takes into account the status text:



At the bottom right of the site, you'll notice some text that gives an indication of what Flash is doing. The leftmost bit of text gives a text message, and the rightmost one gives numeric feedback (such as the percent loaded when the site is online; at the moment it probably won't show anything if you're viewing the site from your local hard drive because load time will be instantaneous).

You'll be adding both of these text areas next.

Choosing a font

The Futuremedia site uses the font Century Gothic because

- Nobody else uses it (always a good reason for a designer!).
- Despite its lack of use within the web design community as a main font, it's actually a very common font, so you shouldn't have too much trouble finding it on the Web if it isn't already installed on your machine.
- Aesthetically, it's a very clean font with a modern feel to it, although it isn't so unsubtle to become another one of those techno font clichés.
- Its uncomplicated appearance has the advantage of making it bandwidth-light when you create a font symbol from it later in the book.

If you don't want to (or can't) use Century Gothic, you can use any sans font of your choice (or simply one that's installed on your machine). Typical choices will include the following:

- Trebuchet
- A font from the Humanist family (be warned that you're unlikely to get good versions of Humanist fonts from the Web without paying for them)
- Arial
- Helvetica

Failing any of these, you can simply use the default Flash sans font, _sans.

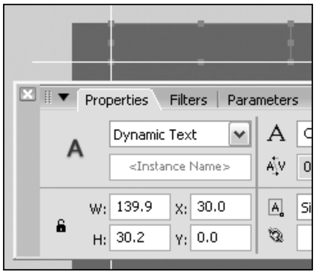
> *You can, of course, use a novel font of your choice. There's a good reworking of the Futuremedia site from a reader (of a previous edition of this book) that uses a handwriting font. The interface has been changed so that it looks like a piece of old parchment. For now, though, let's get the thing working first!*

OK, the preamble is done. Time to get to work.

Adding the text

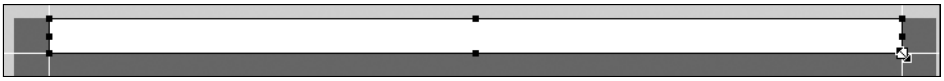
To add the text, follow these steps:

1. Still in the frame layer folder, add two new layers called status and mainTitle.
2. In the layer mainTitle, add a dynamic text field anywhere on the stage. Use the Property inspector to position the text field at an x of 30 and a y of 0 so that it sits at the top of the screen inside the guideline:



Dynamic Text
<Instance Name>
W: 139.9 X: 30.0
H: 30.2 Y: 0.0

3. Now you'll increase the size of the text field. First, double-click the text field to select it and view all the draggable points.
4. Drag the bottom-right square handle all the way over to the right guideline and release it:



It should now cover the majority of the stage, and should sit between the two side guidelines quite snuggly.

5. Inside the text field, enter the text title text. Using the Property inspector, change the text field's attributes and properties as shown in the following table and figure:

Text Type	Dynamic Text
Instance name	main_txt
Font	Century Gothic*
Font size	22
Font color	#999999
Lines	Single Line
Justification	Left

* Or your chosen font

6. This will make your text field go to the top border and extend across it. Lock the mainTitle layer.

> *You may find that some of the values seem to have a life of their own. For example,* 30.0 *may become* 29.9. *If Flash starts adding or subtracting small amounts, subtract or add an equal amount. For example, if you see* 29.9 *instead of* 30.0, *change it to* 30.1. *This usually (but not always) fixes the problem.*

7. Add another two text fields in the status layer, this time clicking the Show Border Around Text icon (in the Property inspector) before you place the text fields. In one text field enter the text ready, and leave the other field blank.

8. Using the Property inspector again, set the following properties and attributes for the text field with ready in it, as shown next:

Text Type	Dynamic Text
Instance name	status_txt
X	505.0
Y	575.0
Font	Century Gothic*
Font size	14
Font color	#999999
Lines	Single Line
Justification	Right

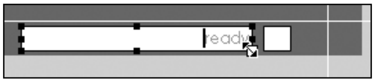
*Or your chosen font

9. Now set the remaining, blank text field as follows:

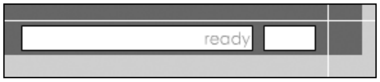
Text Type	Dynamic Text
Instance name	statusValue_txt
X	715.0
Y	575.0
Font	Century Gothic*
Font size	14
Font color	#999999
Lines	Single Line
Justification	Left

* Or your chosen font

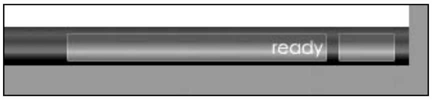
10. Once both text fields are in place, double-click the `status_txt` text field and use the square drag handle to resize it so that it is about 10 pixels short of the `statusValue_txt` text field:



11. Finally, resize the `statusValue_txt` text field in the same way, so that both text fields look like this:



> *Note that the status text currently has a white background. In the final site, the background is a shade of gray that blends in better with the site design (see the following image). The reason you can't set the correct background color just yet is because you can only do it through ActionScript—the Property inspector doesn't allow you to set any background color other than white. Many properties are like this (and, in some cases, entire classes!).*
>
> *You're simply not using Flash to its full potential and configurability until you start using ActionScript.*



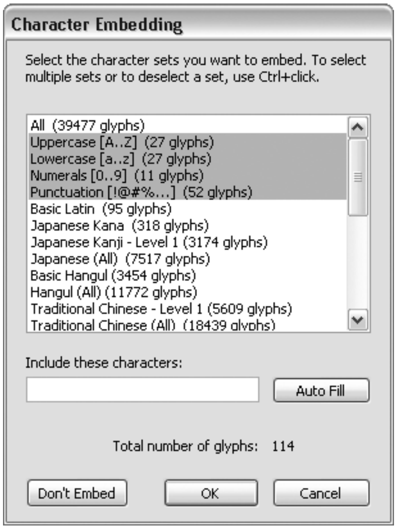
Embedding the font

Embedding a font (or parts of a font) forces Flash to save the font outline data, which allows Flash to treat text using the font as a vector graphic. The advantages of this are as follows:

- The text can be animated in just the same way a movie clip can—it can be scaled and rotated, for example. Text that isn't embedded will disappear if you do anything to it that changes its appearance drastically, because Flash simply doesn't have the information to work out how it should appear once you start altering it at the vector level, unless it has the vector outline data on hand.
- The text will appear with a vector edge. It will be anti-aliased, which means it will appear with a much less pixilated appearance. This is usually a good thing, except when you're using very small font sizes (you aren't in this case).

The Futuremedia site uses the same font throughout. This means you're best off embedding a major part of the font. To do this, follow these steps:

1. Select one of the text fields (it doesn't matter which one because they are all dynamic). In the Property inspector, click the Embed button. The Character Embedding window will appear.
2. In the window that appears, highlight the second to fifth options (Uppercase to Punctuation). Depending on the font you're using, this should embed just over 100 glyphs (individual characters). Click OK to confirm this.



Character Embedding

Select the character sets you want to embed. To select multiple sets or to deselect a set, use Ctrl+click.

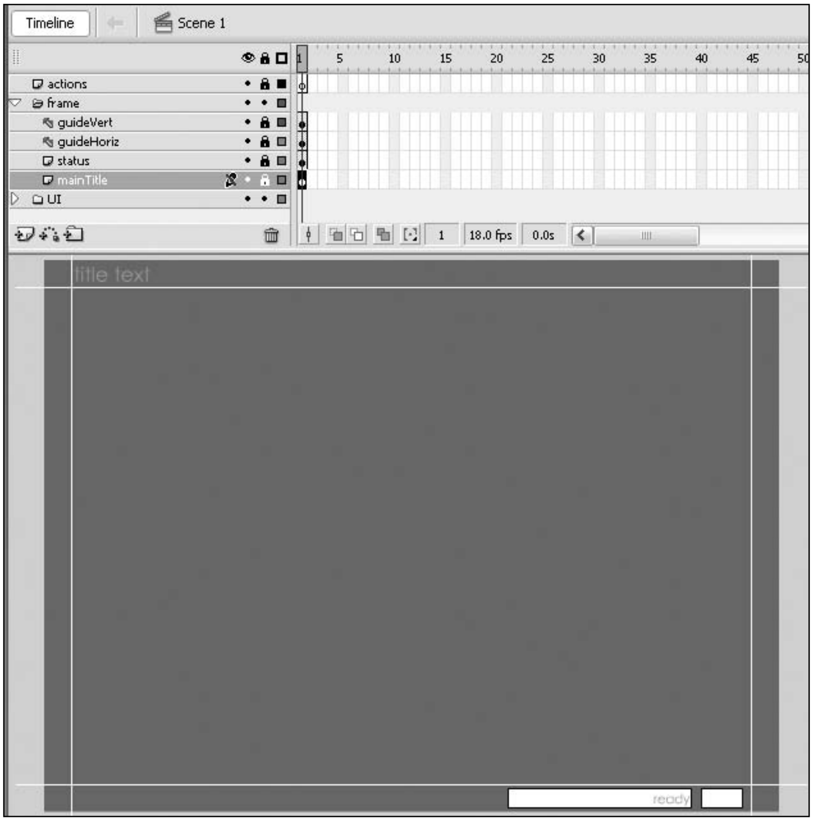
All (39477 glyphs)
Uppercase [A..Z] (27 glyphs)
Lowercase [a..z] (27 glyphs)
Numerals [0..9] (11 glyphs)
Punctuation [!@#%...] (52 glyphs)
Basic Latin (95 glyphs)
Japanese Kana (318 glyphs)
Japanese Kanji - Level 1 (3174 glyphs)
Japanese (All) (7517 glyphs)
Basic Hangul (3454 glyphs)
Hangul (All) (11772 glyphs)
Traditional Chinese - Level 1 (5609 glyphs)
Traditional Chinese (All) (18439 glyphs)

Include these characters:

Total number of glyphs: 114

Parting shots

The site at the moment looks like this:



Not much to look at for sure, but its still early days. The ugly duckling will transform itself into that beautiful swan soon. Don’t worry, it won’t take all winter for this bird . . .

More important than the appearance of this rather drab FLA, you’ll by now have started to appreciate the major difference between working with Flash tweens (as you may have done in this book’s sister work, *Foundation Flash 8*), where you can do everything “by eye,” and the much more precise world of ActionScript, which even seeps into the graphics associated with the scripts.

Summary

In this chapter you’ve looked at how ActionScript uses variables as containers for information you want to use in your movies. You’ve also covered the different types of values that a variable can store:

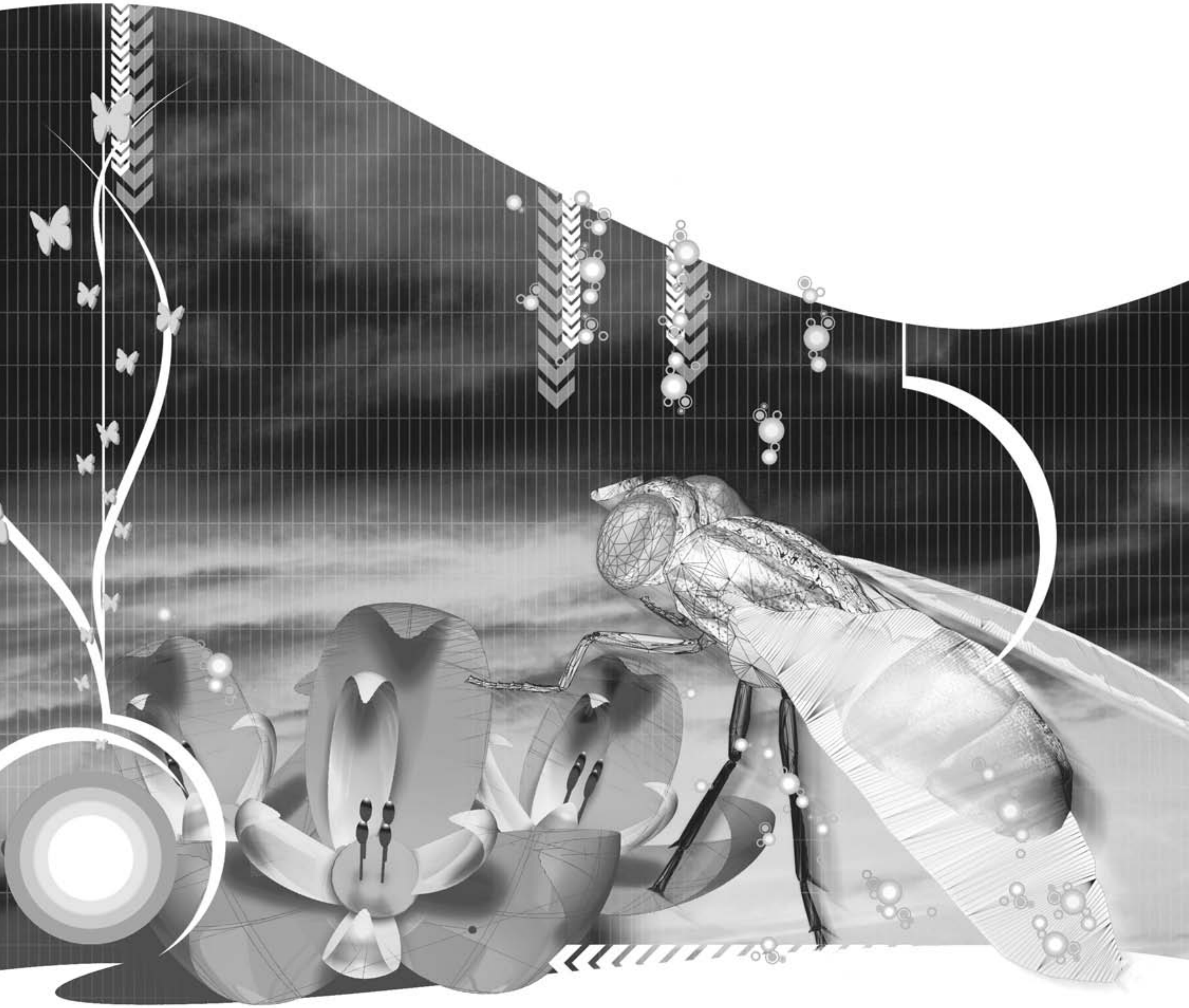
- Strings
- Numbers
- Booleans
- Arrays

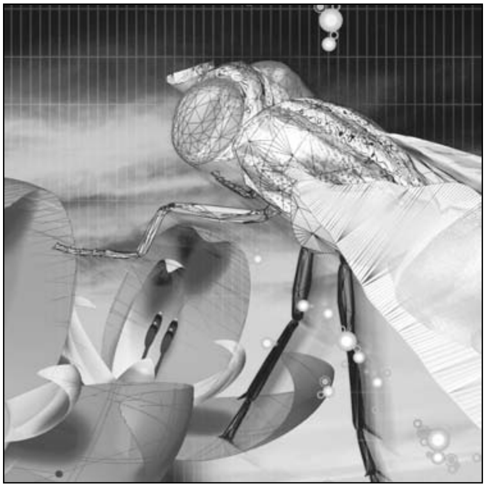
You explored the differences between literal values and expressions, and you saw how to communicate with a user via string values and text fields. We then talked about rules for how you name your variables, so that they’re acceptable to Flash and useful to you. You went on to look at working with numeric values (for performing calculations) and Boolean values (for storing the result of a true/false expression).

Rounding off the chapter, you looked at how arrays make it easy to store lots of related data together in a neatly structured way, which apart from anything else helps you find it easily!

You now have a good grounding in variables and values. Although you didn’t build that many cool FLAs in this chapter, you have a solid foundation to move forward into Chapters 4 and 5, where every new thing you learn will allow you to do something cool and immediately useful. That’s what you’ve built up to by slogging through this chapter!

As you work through the rest of this book, you’ll see how variables lie at the heart of virtually all of the most exciting Flash effects you can create. Have a little break and get ready for Chapter 4.





Chapter 4

MOVIES THAT DECIDE FOR THEMSELVES

What we'll cover in this chapter:

- Getting Flash to look at a situation and decide what to do based on what it sees
- Using sausage notation (or, "How I learned to stop worrying and love braces")
- Giving Flash the intelligence to check passwords

So far, you've looked at a few simple ways to control Flash movies using actions, event handlers, and variables. The actions are like commands, telling Flash what to do. The event handlers are just sets of actions that get triggered by events from the movie so that Flash can respond in real time to things like button presses and entering a new frame. Variables give Flash memory so that you can store information away for when you need it later on.

Already, that gives you quite a lot to play with, but it's not without problems. Think about the last couple of examples from Chapter 3. When you made your "times two" movie, you had to set up the input text field so that it would accept only numbers and the decimal point. Otherwise, it might end up trying to double the value of "platypus," which wouldn't make sense at all!

That's cool, though—you dealt with it. But what about the array of movie titles? It works OK if you put in 1, 2, or 3, but it doesn't give you a bean if you try 4, 5, or 6. Who cares? Well, whoever's using your movie list will care if they're expecting to see something and they see nothing. At least you could put up a sign that says "Sorry, but this array doesn't list movies made after 1963."

This might not seem like much, but it's not something you can do with the tools you've seen so far. What you need is a way to **test** whether the input's good or not, and then **act on the result**. To put it in plain English, Flash needs to **make a decision** about what to do.

Decision making

Real-life decisions are rarely as simple as

I'll go to the open-air rock concert if it's not raining.

You're probably far more likely to think

If I have enough money, and it's not raining, I'll go to the open-air rock concert. But if I get home late from work, I won't have the time.

Your decision now is based not only on whether you'll get wet if you go to the concert, but also on whether you have enough money and can spare the time.

These sorts of issues are just as important when you're making decisions in ActionScript. You need to consider exactly what your conditions are, what should happen if they're met, and what should happen if they're not met.

OK, you may already know what you want to say, but that doesn't mean you know how to state your conditions in a way that Flash is going to understand. Say you write your concert dilemma out like this:

```
if (it's not raining)
  {go to the concert}
```

Er . . . what's with all those brackety things? Well, you've already used parentheses, (and), to pass arguments to actions—gotoAndStop(10), for example. They're information you give to an action to help it do its job. The curly braces, { and }, should look familiar too—you used them around all the event handler actions in Chapter 1. They represent a **code block**, which is simply a set of lines that are executed together.

> *One way to think of this rather complicated syntax is as a string of sausages, with the contents of each set of braces being a sausage. To make each sausage, you have to have a corresponding sausage start and end.*

As you saw, though, the real problem is a little more complex than that:

```
if (I have enough money and it's not raining and I have time)
  {go to the concert}
```

If you were programming this decision, you'd go through exactly this process. Writing a decision down on paper in the same way we've done here will help you to make sure you've considered all the conditions that need to be met.

Making decisions in ActionScript: The if action

So what about making this work for real? It's all very well having this **sausage notation** for working things out, but it's not much help if you can't use it in a movie.

Well, you're actually not so far off from using it in a movie. The **condition** (i.e., the bit you place between the parentheses, (and)) can only ever be true or false. If it's true, you go to the concert. If it's false, you don't. Simple as that.

Now, if you can manage to stretch your mental faculties back to the previous chapter, you may recall looking briefly at Boolean variables. These can hold only one of two values: true or false. What's more, you looked at a couple of ways to generate true and false values, using > (greater than) and < (less than) to test whether one number was bigger or smaller than another.

> *Here's a tip if you're not familiar with greater-than and less-than signs. The small end of the < and > characters always points to the smaller number, and the big end always points to the bigger number when the statement is true. Thus,* a > b *is true if* a *is bigger than* b, *and* a < b *is true if* a *is smaller than* b.

So, once you've worked out an expression that gives you a true or false result, you can store it in a Boolean variable and use that Boolean (or even just the expression itself) as a condition. If the condition is met (in other words, the value is true), then Flash has to do something. If the condition isn't met (the value is false), then Flash doesn't have to do that something.

Let's add that information into sausage notation. Assume you've already defined some Boolean variables like this:

```
var haveEnoughMoney:Boolean = true;
var notRaining:Boolean = true;
var haveTheTime:Boolean = true;
```

You know that && is something you can use to check whether one Boolean **AND** another are both true. Handily, you can even string several together to check whether one AND another AND another are all true, so this is what you could end up with:

```
if (haveEnoughMoney && notRaining && haveTheTime)
  {go to the concert}
```

If any of the Booleans are set to `false`, you'll be staying in!

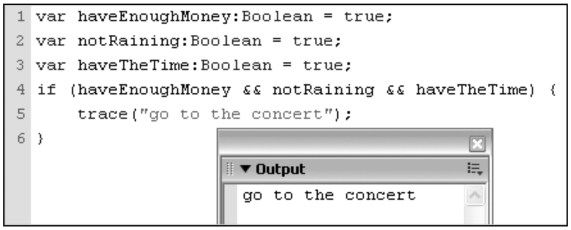
What about telling Flash to go to the concert? How does that shape up ActionScript-wise? That's going to depend on what your movie actually does. Maybe you're going to send a little stick man running up to a big stage, where he joins the screaming masses. Or maybe you're just going to make a little message appear that says "go to the concert". Well, the first one would look more impressive, but the second one would be quicker!

If you want to make your own movie that shows a man running up to join the crowd, then you can write actions to make it do that. The important thing to remember is that you can put **any actions you like** in here, and Flash will perform them for you, but **only if the condition is true**.

So how do you make this into proper ActionScript that a movie's going to understand? You don't have to change a lot. In fact, if you type it in exactly as it stands, it should work just fine. However, to keep the syntax checker happy, you need to rearrange it ever so slightly:

```
var haveEnoughMoney:Boolean = true;
var notRaining:Boolean = true;
var haveTheTime:Boolean = true;
if (haveEnoughMoney && notRaining && haveTheTime) {
    trace("go to the concert");
}
```

Just put the opening brace { on the first line, indent the action (to remind you that it's **nested** inside the `if` statement), and put the closing brace } on a line of its own. Presto!



```
var haveEnoughMoney:Boolean = true;
var notRaining:Boolean = true;
var haveTheTime:Boolean = true;
if (haveEnoughMoney && notRaining && haveTheTime) {
    trace("go to the concert");
}
```

```
go to the concert
```

Notice that you don't put a semicolon after the closing brace. The only time you need a semicolon after a closing brace is when declaring an anonymous function (like the functions attached to event handlers in Chapter 1). Functions will be described in more detail in Chapter 6.

If you try this in a new Flash document and it doesn't work, just remember that ActionScript 2.0 is **case-sensitive**, meaning that ActionScript treats notRaining and notraining as completely different variables—and your code will fail if you mix them up. The names of the data types (such as Boolean) are also case-sensitive; they all begin with an initial capital.

You can see the effects of changing your variables by making one or more of them false. Note that you **won't** get a "don't go to the concert" message; instead, you'll get no message at all.

So, in general, an if statement looks like this:

```
if (condition is true) {
  do this;
  do this;
  do this;
  ...
}
```

the code block (the lines of code within the { }) will run only if the condition is true. If the condition is false, ActionScript silently ignores it and passes on to whatever comes after the code block. If nothing comes after the block, the script stops running.

The only structural difference between this and sausage notation is that ActionScript prefers you to split the sausage itself over several lines.

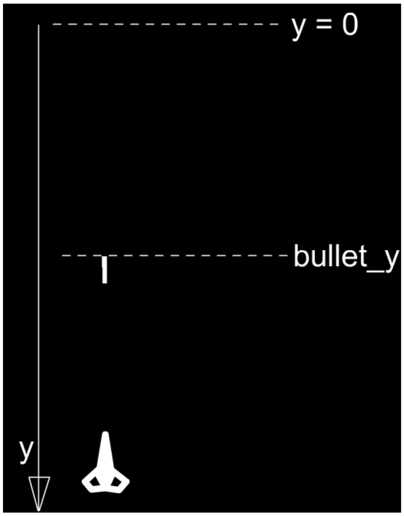
Sausage notation will help you to write all your programming decision branches in terms that you can use to set things clearly in your mind before you start programming. As an added bonus, it's written in correct ActionScript syntax, so you can convert it seamlessly into working code. Don't be put off by the silly "sausage" name—it really does help!

Now that you have the basic idea, let's apply this decision-making system to the kind of situation you might encounter when you're thinking out a Flash movie.

Defining a decision

You'll start by expressing your decision as simply and precisely as possible, in order to get a clear idea of what you want the code to do.

Say you're writing a game of space invaders in which the player controls a ship that can fire bullets up at the attacking alien hordes. In particular, you're considering the situation when the bullet has been fired (signified by `bulletFired = true`) and is moving up the screen (at a speed controlled by the numeric value of `bulletSpeed`). If it hasn't already reached the top of the screen (where the y-coordinate is 0), you want to move it further up. If it's **at** the top, you want to stop moving the bullet, remove it from the screen, and set `bulletFired` to `false`, indicating that the player is allowed to fire again. The value of variable `bullet_y` denotes the bullet's height at any point in time.



When you're faced with such a longwinded problem, there's only one way to simplify it: lose some of the words. The easiest way to do this is to draw a picture—like the one alongside—which automatically drops your word count to zero.

The player's ship is at the bottom of the screen, and the bullet is at some point above it, moving upward. The bullet will always move up in a vertical line, so only its y-coordinate will change, decreasing as it moves up the screen until it reaches the top and has a y-coordinate of 0.

You're now concerned with building a framework of decisions that will result in this behavior. You need to consider two questions:

1. How does the bullet behave when it's onscreen?
2. How does the bullet behave when it first goes off the top of the screen?

`bulletSpeed` is a variable that you'll use to hold a number that will tell Flash how many pixels to move the bullet up the screen. Each time the script runs, this value will be subtracted from the bullet's y-coordinate position, moving it to a new position. Making it large will make the bullet move quickly; making it small will make the bullet move slowly.

The y-coordinate of the bullet, `bullet_y`, starts off as a positive value and reaches 0 when it gets to the top of the screen. In other words, `bullet_y` will be positive before it hits the top and negative once it's gone off the screen.

- If `bullet_y` is positive, the bullet is onscreen and you should keep subtracting `bulletSpeed` so that the bullet moves upward.
- As soon as `bullet_y` is less than 0, you no longer need to animate the bullet, although you need it to disappear. You could make it disappear in a number of ways, but the easiest is to place it offscreen, say at `bullet_y = -100`. Also, once the bullet is off the screen, the player should be allowed to fire again, so you need to tell Flash this by making `bulletFired = false`.

So, your decision making breaks down into two branches. This is how they look in sausage notation:

```
if (the bullet has been fired and its y-coordinate is greater than 0)
  {move bullet}
```

and

```
if (the bullet has been fired and its y-coordinate is less than 0)
  {hide bullet and let user fire another}
```

So what happens if the bullet hasn't been fired? Well, neither of these branches will be relevant, and neither will do anything. In fact, that's just what you want: if `bulletFired` is `false`, the bullet should be safely hidden away above the top of the screen, and that's just where you want it to stay!

Let's put some variables into sausage notation:

```
if (bulletFired is true and bullet_y is greater than 0)
  {set bullet_y equal to bullet_y minus bulletSpeed}
```

and

```
if (bulletFired is true and bullet_y is less than 0)
  {set bulletFired equal to false and set bullet_y to -100}
```

Now you just need to convert this into proper ActionScript syntax. This is easy enough to do:

```
if (bulletFired && bullet_y > 0) {
  bullet_y = bullet_y - bulletSpeed;
}
```

and

```
if (bulletFired && bullet_y < 0) {
  bulletFired = false;
  bullet_y = -100;
}
```

There are two things to note about how we've written out the conditions here:

- Each set of conditions is completely contained within the parentheses ( ). When you start using more complex conditions (with their own subterms in parentheses), it's all too easy to forget that **the condition as a whole** needs to be contained.
- We've replaced the subcondition *`bulletFired is true`* with `bulletFired`.

You may want to think about this second point for a moment. Just as you can use < and > to test whether one value is bigger or smaller, you can use == to test whether two values are the same. So why not do the obvious thing and replace *`bulletFired is true`* with `bulletFired == true`?

Well, `bulletFired == true` would only ever be true if `bulletFired` itself was true. Likewise, it would only ever be false if `bulletFired` itself were false. In other words, a Boolean variable like `bulletFired` can **always** be used on its own as a true/false test. There's nothing wrong with using `bulletFired == true`, but it should be fairly obvious which one looks neatest!

One more thing to consider is that the y-coordinate of –100 (which you use to put the bullet well clear of the visible screen) is a fairly arbitrary value. It's obvious what it does now, but six months from now you might find yourself asking, "What's that –100 doing in there?"

The obvious way to make things clearer is to add a comment by this line. Another way to clarify is by using a descriptively named variable, such as `HIDE`. *At the beginning of the game you could set* `HIDE` *to –100, and then in the main code you could make statements like* `bullet_y = HIDE`, *which makes what this value represents explicit. The fact that a variable name is in CAPITAL LETTERS also implies that it's a constant value—one that won't change.*

Password protection part 1: Setting up the stage

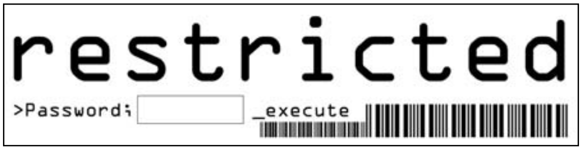
Let's walk through a simple example and put all this theory into practice. We'll work with a techno-gothic screen whose sole purpose is to extract a password from the user. If the user fails at this point, he or she won't be allowed to enter the rest of the site.

It's important to note that the exercises in this chapter are designed to demonstrate how to work with conditions in ActionScript. They are not *intended for use in a system that needs to be secure. Don't think for a moment that they will lock ne'er-do-well hackers out of your site for more than ten seconds!*

1. Open `password_start.fla` from this chapter's download bundle. It consists of some basic graphics, a text field, and a button. What this movie **doesn't** have is the ActionScript required to make it run—that's where you come in!

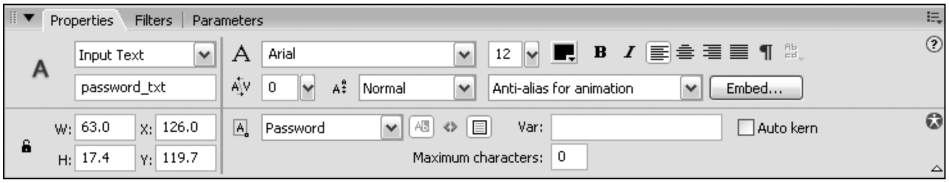
Font freaks will probably want to know the name of the font used in this site. It's called OCR-Extended. For the download files, we've applied Modify ➤ Break Apart *a couple of times (breaking all type into shapes) and then grouped them together for ease of use. You should therefore be able to see the screens just as they were designed, even if you don't have this specific font installed.*

2. Select frame 1 in the timeline, and you'll find the following screen (which is defined on the graphics layer):



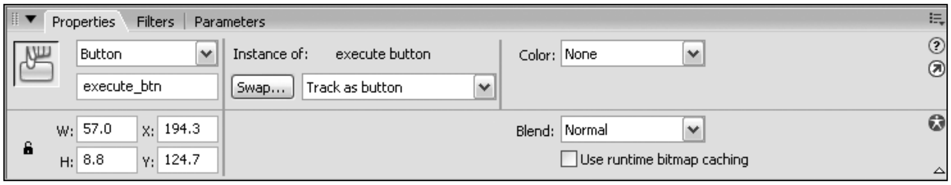
There are two active items on this screen. The hollow rectangle is an input text field, where a user can enter a password. The text that reads _execute is actually a button, which the user will click after entering the password. Right now, neither one is configured, so that's the first thing you need to change.

3. Select the input text field and use the Property inspector to configure it like the following one. In particular, set the instance name to password_txt, set the line type drop-down to Password (so that Flash displays only asterisks in the text field when you type into it), and click the Show Border Around Text icon (so that the user knows where the text field is).



Input Text
password_txt
Arial
12
0
Normal
Anti-alias for animation
Embed...
W: 63.0 X: 126.0
H: 17.4 Y: 119.7
Password
Var:
□ Auto kern
Maximum characters: 0

4. Now select the button and set its instance name to execute_btn.



Button
execute_btn
Instance of: execute button
Swap...
Track as button
Color: None
W: 57.0 X: 194.3
H: 8.8 Y: 124.7
Blend: Normal
□ Use runtime bitmap caching

The stage is all set up now, so you just need to wire it up to some ActionScript so that your active items can actually do something. Let's pause for a moment and consider what you're trying to achieve.

Password protection part 2: Working out the ActionScript

Once the user has entered a valid password, we want to take him or her to the next stage of the movie, which will start at frame 21. We've used the Property inspector to label this frame as accepted. This is the point where the site proper would begin, and this is what the user will see:

Before you let the user get this far, though, you want to check whether he or she has used the correct password. This needs to be checked just as soon as the user clicks the _execute button in frame 1, so that's where you need to attach the ActionScript that will make the system work. Let's work out that ActionScript now.

You need to make Flash look at what input the user gives you via the contents of `password_txt`, and compare it with a predefined password. If they're the same, Flash will let the user into the site.

In recognition of a great film, let's make the password "I am the One". Also, to make life easier for the user, let's make the password case-insensitive, so the user can enter I Am The One, or I am the one, or any combination of uppercase and lowercase letters.

Here is the decision expressed in shorthand:

```
if (lowercase version of the text in password_txt is "i am the one")
  {go to the "accepted" frame}
```

Irrespective of what the user inputs, the text will be converted to lowercase before the comparison is made, so it doesn't matter if the user types I am the One *or* I AM THE ONE. *The user will still be "the One" either way.*

Note that this doesn't consider what will happen if the password provided isn't a good match. In that event (for now, at least) you don't want to do anything. In ActionScript, you need to write the following:

```
var lowerCasePass:String = "";
lowerCasePass = password_txt.text.toLowerCase();
if (lowerCasePass == "i am the one") {
  gotoAndStop("accepted");
}
```

The first line initializes a variable called `lowerCasePass` and declares its type as `String`. The following line set its value to a lowercase version of the contents of `password_txt` (`password_txt.text`), which you get by sticking `.toLowerCase()` onto the end of it. At this stage, you don't really need to worry about **how** this works; just remember **what** it does and that the case of the letters the user enters now doesn't matter.

The next three lines of code make up the decision-making system using this structure:

```
if (something) {
  do this;
}
```

Note that the `something` expression in the actual code uses the double equal sign (==) or **equality operator**, which tests whether the terms on either side have the same value. It will return `true` only if `lowerCasePass` has exactly the same value as the string `"i am the one"`. If that **is** the case, the `do this` section will be executed, so the movie will jump to the frame labeled accepted.

A common beginner's mistake is to use a single equal sign instead of the equality operator. If you use a single equal sign, you end up assigning to the variable the value on the right, rather than comparing the two. Since they are both of the same data type, the assignment will succeed and equate to true. So it wouldn't matter what the user enters, it would always succeed. It's an easy mistake to make, so if you find you're getting strange results from comparisons, check that you're using the double equal sign and not a single one.

You don't want this script to be executed as soon as the movie starts—after all, you need to give the user a chance to enter a password first! Since you want it to run just after the `execute_btn` button has been clicked, it makes sense to put it into a function to be triggered by that button's `onRelease` event. Basically, you're going to turn the code into an event handler and attach it to a callback for the `execute_btn.onRelease` event:

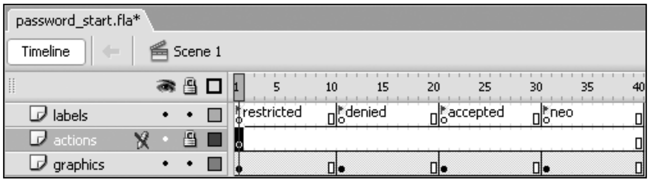
```
execute_btn.onRelease = function() {
  lowerCasePass = password_txt.text.toLowerCase();
  if (lowerCasePass == "i am the one") {
    gotoAndStop("accepted");
  }
};
var lowerCasePass:String = "";
stop();
```

Note that you also have to tell the movie to stop, otherwise it would run straight on to the next page, whether or not the user entered a password, valid or not! You also move the definition of the variable `lowerCasePass` to outside the event handler so that it can be used in other parts of the script, if required, later. It may seem odd to put the variable definition after the event handler function, but there is a good reason for this, which will be explained later in this chapter.

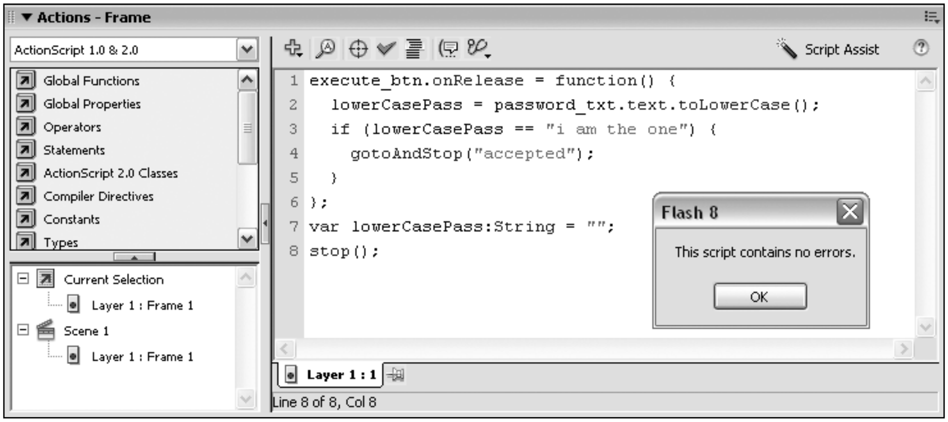
Password protection part 3: Wiring up the code

Now you're ready to plug this code into the movie and give it a test run.

1. Use the Timeline panel to select the first frame in the actions layer.



2. Now open the Actions panel, type in the code, and click the Check syntax button (it's the fourth one from the left at the top of the Actions panel and looks like a check mark or tick) to confirm that it's good to go.



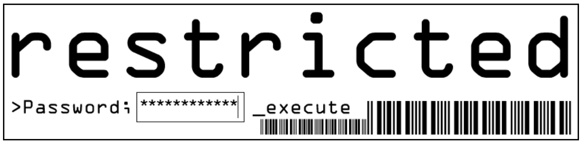
```
execute_btn.onRelease = function() {
  lowerCasePass = password_txt.text.toLowerCase();
  if (lowerCasePass == "i am the one") {
    gotoAndStop("accepted");
  }
};
var lowerCasePass:String = "";
stop();
```

Flash 8

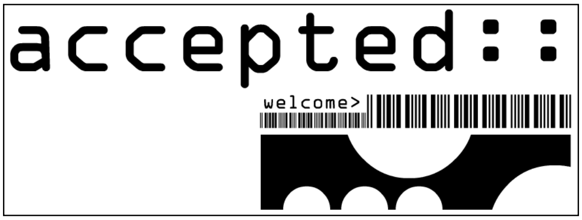
This script contains no errors.

Line 8 of 8, Col 8

3. You're now ready to test the movie. Choose Control ➤ Test Movie or press Ctrl+Enter/Cmd+Return. Type the password into the input text field. (You'll see each character represented by an asterisk or blob, so that no one can see what you've typed. That's why we told you to select Password from the Property inspector for the text type.)



4. Now click the _execute button. Assuming you typed in the password correctly, you'll now see the accepted message shown here:



Alternative actions

So what happens if you enter the wrong password? The answer: nothing at all. In fact, until you key in a valid password, it will look for all intents and purposes as though the _execute button is broken! If you've ever come across this kind of behavior, you know how frustrating it can be—you've typed in your password, but the movie/website/application still won't run properly and doesn't even hint at what's wrong.

Ideally, you should give the user some feedback when his or her password is invalid, and that means taking another look at your decision-making code. Here's the important bit as it stands, first in sausage notation, and then in ActionScript:

```
if (the text in lowerCasePass is "i am the one")
  {go to the "accepted" frame}

if (lowerCasePass == "i am the one") {
  gotoAndStop("accepted");
}
```

The most obvious way to deal with an invalid password is to add another `if` structure on the end, like this:

```
if (the text in lowerCasePass is "i am the one")
  {go to the "accepted" frame}
if (the text in lowerCasePass is not "i am the one")
  {go to the "denied" frame}

if (lowerCasePass == "i am the one") {
  gotoAndStop("accepted");
}
if (lowerCasePass != "i am the one") {
  gotoAndStop("denied");
}
```

That's another new symbol for you: just as `==` means "is equal to," `!=` means "is not equal to." So, in the second `if` statement, the condition is `true` only if the text is **not equal** to your password. This works OK, but it's a bit of a waste of time—you're basically spelling out the same condition twice. It would be much easier just to write this:

```
if (the text in lowerCasePass is "i am the one")
  {go to the "accepted" frame}
otherwise
  {go to the "denied" frame}
```

Don't go looking for the `otherwise` action, though, because it doesn't exist. ActionScript uses `else` to do this:

```
if (lowerCasePass == "i am the one") {
  gotoAndStop("accepted");
} else {
  gotoAndStop("denied");
}
```

Acting on alternatives: The else action

The else action tells Flash to execute the attached commands whenever the if condition turns out to be false. The code you've just seen is a good example of the commonest sort of choice you'll find in programming. You have a condition with two possible outcomes (in this case, the password is either valid or it isn't), and you define an appropriate course of action for each one.

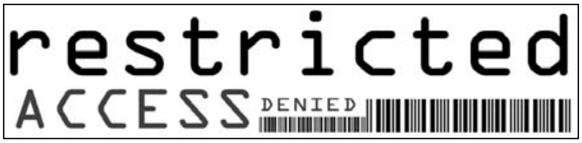
Access denied: Using the else action

Let's add this useful little nugget into the password movie.

1. Use the Timeline panel to select the first frame in the actions layer again.
2. Add the new lines of code (highlighted here in bold) into the Script pane of Actions panel, and click the Check syntax button to check the syntax (refer back to step 2 of the previous exercise if you need reminding of where to find it).

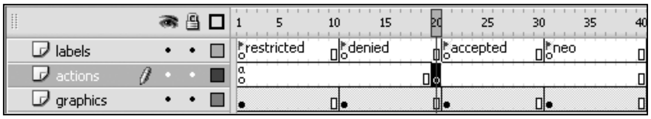
```
execute_btn.onRelease = function() {
  lowerCasePass = password_txt.text.toLowerCase();
  if (lowerCasePass == "i am the one") {
    gotoAndStop("accepted");
  } else {
    gotoAndStop("denied");
  }
};
var lowerCasePass:String = "";
stop();
```

3. You can now press Ctrl+Enter/Cmd+Return to test run the movie again.
4. Try clicking the _execute button when there's no text in the password box. You should now see the access denied message like this:



Great! That's just what you wanted. The only problem now is that you can't get back to the original screen to try again. Let's fix that right away.

5. Select frame 20 on the actions layer (see the following image). This is the last frame containing the access denied message. Press F6 to create a keyframe here.

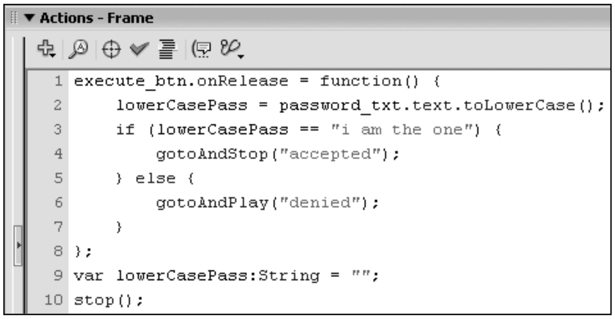


6. Now open the Actions panel and attach the following code to the new keyframe:

   ```
   gotoAndStop("restricted");
   ```

   As soon as the playhead reaches frame 20, this action will send you back to the original screen.

7. This is no good, though, unless frame 20 actually gets played, so you need to make a slight change to the original code. Specifically, `gotoAndStop("denied")` needs to become `gotoAndPlay("denied")`. Select frame 1 in the actions layer and make this change so that your Actions panel looks like the screenshot (we've also closed the panes to the left of the Script pane by clicking the little left-facing arrow at the left side of the pane).

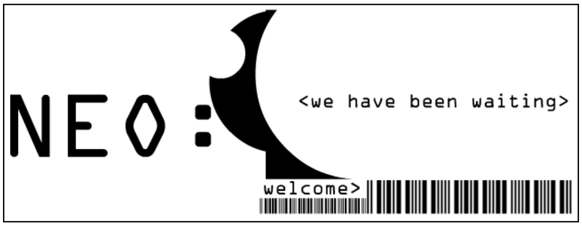


```
execute_btn.onRelease = function() {
    lowerCasePass = password_txt.text.toLowerCase();
    if (lowerCasePass == "i am the one") {
        gotoAndStop("accepted");
    } else {
        gotoAndPlay("denied");
    }
};
var lowerCasePass:String = "";
stop();
```

8. Press Ctrl+Enter/Cmd+Return, and test the movie with no password again. This time, you should find that the access denied message is shown for less than a second before it returns you to the password entry screen. Problem solved!

More than one alternative: The else if action

Let's think about how you might improve this system even further. What if you wanted to create a special section of the site for executive users, to whom you've given a different password? If you take a look at frames 31 through 40, you'll see that we've already made the following screen (labeled neo) to welcome these VIPs:



You'll also need to pick a separate, executive password (we've chosen the word "Neo") and some more ActionScript to check whether it's been entered. Let's use some more sausage notation to confirm what you're doing:

```
if (lowerCasePass is "i am the one")
  {go to "accepted" frame}
otherwise if (lowerCasePass is "neo")
  {go to "neo" frame}
otherwise
  {go to "denied" frame}
```

You might have guessed from this that the ActionScript you need to make this happen looks like this:

```
if (lowerCasePass == "i am the one") {
  gotoAndStop("accepted");
} else if (lowerCasePass == "neo") {
  gotoAndStop("neo");
} else {
  gotoAndPlay("denied");
}
```

You effectively combine the `else` and `if` actions so that you can test for another condition if the first one isn't met. Of course, if neither one is met, you'll end up running the `else` command just like before. All it really means is "if the previous condition tested was false, check if this one's true."

In fact, ActionScript treats `else if` as an action in its own right, and there are things you can do with an `else if` that you can't do with a plain `else`. For example, you can put as many `else if` actions as you want after an initial `if` action, making Flash check if any number of conditions have been met (are true):

```
if (a) {
  do this;
} else if (b) {
  do this;
} else if (c) {
  do this;
} else if (d) {
  do this;
}
```

Breaking this down a little, consider the following:

```
if (a) {
  do this;
} else if (b) {
```

Flash will start by looking at the first condition, a. If it's true, Flash will carry out the associated *{do this;}* action (or series of actions) and then skip past the rest of the `if...else if...else` block. If a is false, though, Flash moves straight on to the next `else if`, which tells it to check if b is true.

Then, if b is true, Flash runs the associated *{do this;}* action and skips to the end of the block. It won't check c or d, and it won't run c or d's *{do this;}* action, **even if** c **or** d **is true**.

If, on the other hand, b is false, Flash moves on to c, and so on, and so on. You need to always remember that **only one** of your if/else if/else commands will run, and that it will be the first one in the list whose condition is true.

> *This is an essential point to understand, because it shows the importance of the order in which you enter your* else if *actions. If several conditions are satisfied, only the first one in the list will ever be acted on.*

So, why not just use a string of separate if structures? Well, it's largely a matter of efficiency: once a condition within an else if command has been found to be true, Flash won't bother checking any of the subsequent else if conditions—it just follows the appropriate instruction. If you used separate if actions, you'd force Flash to check every single one, even after it had found a condition that had been met. If you want only one result, this is going to put a completely unnecessary strain on Flash and possibly slow down your movie.

Essentially, as you go down the ladder of else if actions, you're finding out what **isn't** true and then homing in on what **is** true by asking more and more precise questions based on what you know it **might be**.

Imagine that you're writing a piece of code that controls how a diving space invader attacks the player on its way down the screen, and you want to make your space invader do different things as it gets closer to the player's ship. You could write your decision like this:

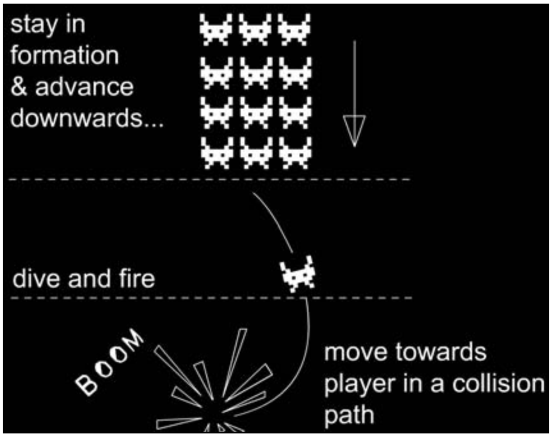
```
if (distance is greater than 50)
  {make alien stay in formation}
else if (distance is greater than 25)
  {make alien fire at ship and dive in a random downward direction}
else if (distance is greater than 10)
  {execute kamikaze tactics}
```

If the first condition in the if action is found to be false, you can deduce that on the first else if statement the distance is less than 50, because if it wasn't you wouldn't have gotten this far down the statement. If the condition in the first else if statement is found to be true, you know that the distance must be greater than 25 but less than 50.

> *Note that the ordering of the* if...else if *decisions affects the order in which they're checked. If you put the "greater than 25" decision after the "greater than 10" decision, the "greater than 25" bit would never run!*

So, to make your decision, you're using what you know the distance **isn't** as well as what it **is**.

As the demarcation lines used to decide the kamikaze behavior move up the screen, the aliens' scope for attacking will increase as they acquire more room in which to perform their kamikaze maneuvers. So, if you gradually moved the lines upward as the game progressed, your aliens would appear to grow more aggressive over time. It's amazing how much an object's behavior can be modified by changing a single number.


stay in formation & advance downwards...
dive and fire
BOOM
move towards player in a collision path

As you can see from this explanation, `else if` has many more versatile uses than you're exploiting in this simple password exercise.

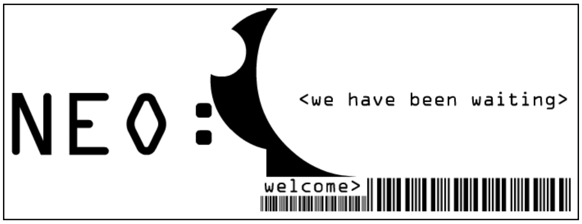
Speaking of that, let's get back to it now. You'll plug in your `else if` action so that Flash can distinguish between the two passwords and direct users to the appropriate part of the site.

Executive access: Using else if

1. Continue working with the same FLA as before. Select the first frame in the actions layer and add the new lines of code (highlighted in the next listing) to what's already shown in the Script pane of the Actions panel. Remember that the second line converts all passwords to lowercase; this is why Neo has become neo. Click the Check syntax button to check the syntax.

```
execute_btn.onRelease = function() {
  lowerCasePass = password_txt.text.toLowerCase();
  if (lowerCasePass == "i am the one") {
    gotoAndStop("accepted");
  } else if (lowerCasePass == "neo") {
    gotoAndStop("neo");
  } else {
    gotoAndPlay("denied");
  }
};
var lowerCasePass:String = "";
stop();
```

2. Press Ctrl+Enter/Cmd+Return to test-run the movie again, and try the new password.
3. Click the _execute button and you should see the neo message like this:



You can find this complete version saved as `password_finished.fla` in the chapter download.

You've now looked at every stage of the `if...else if...else` cycle and considered the decisions that Flash makes at each point along the way. Hopefully, you're now starting to get an idea for just how much flexibility these decision-making structures can give your Flash movies. They don't just act and react—they can take a different course of action for each circumstance you plan for.

There's one more way to make decisions using ActionScript. While it doesn't actually do anything new, this method can be a lot tidier (not to mention shorter and more readable) than the `if...else if...else` cycle you've used so far.

Handling lots of alternatives: The switch action

When you last saw the password script, the meat of it looked like this:

```
lowerCasePass = password_txt.text.toLowerCase();
if (lowerCasePass == "i am the one") {
  gotoAndStop("accepted");
} else if (lowerCasePass == "neo") {
  gotoAndStop("neo");
} else {
  gotoAndPlay("denied");
}
```

It doesn't exactly take a degree in rocket science to guess what this does. Take out all the parentheses and braces, swap "is equal to" for the == symbols, swap "otherwise" for else, and stick it all on one line. Bingo! You've got something that's pretty close to plain English:

- If `lowerCasePass` is equal to "i am the one", go to accepted and stop there.
- Otherwise, if `lowerCasePass` is equal to "neo", go to neo and stop there.
- Otherwise, go to denied and play from there.

That's perfect for this situation: it's simple, concise, and easy to understand. Believe it or not, rule number one of good programming says that **all** code should aim to be these three things. Maybe someone should tell the programmers that!

In real life, though, you might want to add more users to the system, each with a unique password, and each with an individual set of actions to perform once the password has been entered. In that case, you could end up with a great long string of `else if (lowerCasePass == "something")` terms. That's not so bad in itself, but it does mean that you're testing the value of `lowerCasePass` over and over and over.

When that happens, it's worth considering using the `switch` action instead. Let's take a look at the `if...else if...else` password testing script when it's written out using `switch`:

```
switch (lowerCasePass) {
case "i am the one" :
  gotoAndStop("accepted");
  break;
case "neo" :
  gotoAndStop("neo");
  break;
default:
  gotoAndPlay("denied");
}
```

Hmm. It's a bit further away from plain English than `if` and `else`. So what does it mean?

Let's take it one line at a time:

```
switch (lowerCasePass) {
```

The `switch()` action is very similar to `if`, in that it uses a sausagelike arrangement of parentheses and curly braces. The big difference here is that you don't put a condition (something that will give you a true or false value) inside the parentheses. Instead, you put in a variable on its own—in this case, `switch (lowerCasePass)`.

So how does Flash know whether or not to run the actions inside the curly braces? The answer is that Flash doesn't need to: **it runs the actions in the curly braces anyway**.

OK, you're shaking your head now. Supper's ready, and this is all getting a bit strange. Just bear with us a moment longer, because this is the clever bit that will one day make your ActionScripting chores seem a hundred times easier!

When you take a look at what's inside the braces, you'll see it doesn't look like anything you've seen before. There are a couple of new terms, `case` and `break`, plus colons (`:`) dotted around all over the place. What does it all mean?

Well, the first thing Flash sees when it starts running the actions is this line:

```
case "i am the one" :
```

This line basically says, “Check whether the string 'i am the one' is the same as what’s being stored in the switch variable.” In this case, the switch variable is lowerCasePass. If the string is the same as what’s stored in the switch variable, then Flash runs the actions following the colon.

> *This* case *thing isn’t actually an action, but a special keyword that helps the* switch *action to do its job. This might not seem like much of a distinction, but it’s important to remember. If you try using* case *outside the curly braces of a* switch *statement, Flash won’t understand what you mean.*

Let’s look at what Flash sees when it starts going through those actions. First up:

```
gotoAndStop("accepted");
```

That’s safe enough. You know what that does already. What about the next line, though?

```
break;
```

This action simply tells Flash to ignore the rest of the actions inside the curly braces. If the switch variable lowerCasePass **does** contain the string "i am the one", then you’re all done now.

What if it doesn’t then? Well, you continue through the contents of the curly braces.

No prizes for guessing what happens if the switch variable contains the string "neo"! The next line primes Flash for precisely that situation:

```
case "neo" :
```

and then follows up with actions to run in that case:

```
gotoAndStop("neo");
break;
```

Finally, you set up a default case:

```
default :
  gotoAndPlay("denied");
```

Note that you don’t need to use the case keyword here. You can think of default as being a bit like case’s poverty-stricken cousin: they’re both keywords that mean something only when they’re used inside a switch statement. But while case is very fussy about the value of the switch variable, default sits at the bottom of the pile and grabs anything and everything that case lets through to it.

It’s quite tempting to describe this default section as “what happens if none of the other cases work out.” That’s certainly what happens in this particular example; if either of the cases were good, you’d have hit a break by now and wouldn’t even be considering the default case.

But what if you take out break from each case statement?

```
switch (lowerCasePass) {
case "i am the one" :
  gotoAndStop("accepted");
case "neo" :
  gotoAndStop("neo");
default :
  gotoAndPlay("denied");
}
```

Zoinks! Even shorter than before, but is it actually what you want?

Say you have the value "i am the one" stored in lowerCasePass. You hit the first case statement, it sees a good match, and it gives you the thumbs up to run the subsequent actions. The playhead gets told to jump ahead to the accepted frame and stop there, but the ActionScript continues running.

When you hit the next case statement, Flash doesn't bother checking! You're already past security—and it really can't be bothered to throw you out now—so the fact that you're not "neo" doesn't stop you from seeing the playhead ordered to the neo frame.

That's pretty bad news, but there's worse to come! Flash is not content with letting you into the executive area of the site on the strength of a standard password. When you hit the default case, it will throw you out of the site regardless!

OK, you've now established that's **not** what you want. But that's not to say you should never consider taking out the breaks—it's just that in this case you really do need them.

Before we move on from switch*, it's worth mentioning that the* switch/case/default *structure is actually converted to* if *structures in the final SWF—Flash Player doesn't know anything about* switch *because it never sees it.*

The switch *structure is more compact and easier to read than the same thing stated in* if*, especially if you have lots of nested decisions to be made (as you'll see in the next example). The downside to using* switch *is that the* if *structure is closer to the final compiled code that will live in the SWF, and it's easier to create optimized code if you use only* ifs*—something that you'll want to do if you're writing processor-intensive applications like Flash games.*

Before we finish off the chapter, let's look at another example of decision making in a Flash movie in which it **does** make sense to use a switch statement that doesn't have any breaks.

Switching without breaks

Just for a few pages, let's go back to school. Say you're creating a Flash movie that's going to let members of a college faculty access records. You can assume there are several different member types to allow for; for simplicity's sake, you'll just consider students, professors, and the head of the faculty.

- Students will want a link to timetables, lab allocations, and so on.
- Professors want a link to all the things students have access to, plus hall bookings, staff allocations, and so on.
- The faculty head wants all of the above, plus a detailed rundown of the faculty's financial status (and maybe links to stock prices for the companies all the faculty money has poured into over the years . . .).

The important thing about this situation is that the information you give to each group isn't always **exclusive** to that group. In the password-handling movie, you always wanted to send the user to one frame **or** another, **or** another, but never to more than one.

In this example, you're dealing with basic faculty information about stuff like lecture timetables, and you want to make that information available to **everyone**. Then there's staff administration stuff that everyone **except** the students needs to know about. Finally, there's key financial information that the head of faculty wants to keep private.

If you tried scripting this with an `if...else if...else` structure, it would look a bit of a mess:

```
if (user is a student)
  {show student info }
else if (user is a professor)
  {show staff info, show student info}
else if (user is the faculty head)
  {show accounts, show staff info, show student info}
else
  {tell user that this site is not for them}
```

It will work, but there's lots of pointless repetition in there. Just for a start, there are three lots of `show student info`, each one of which might consist of **loads** of separate actions.

The answer is to use a `switch` structure, just like the one you used before, but without any breaks:

```
switch (type of user) {
case "head of faculty" :
  show accounts;
case "professor" :
  show staff info;
case "student" :
  show student info;
default :
  tell user that this site is not for them;
}
```

You start by finding out what type of user you're dealing with. For the sake of keeping the example simple and flexible, assume that this is something the user just types in.

> *Of course, in practice that's the last thing you'd want to do, since any old John Doe typing in* head of faculty *could then sneak a peek at privileged information! While security is an important issue to be aware of, it's a bit of a red herring while you're still on the learning slopes—that is, unless you're planning to try to base a real "secure" site on these very simple examples, which we wouldn't recommend for a millisecond!*

If the user claims to be the head of faculty, the first `case` statement shows the user a link to the faculty accounts page. If not, the user doesn't see the accounts.

Either way, you then move on to the second `case` statement. If you're dealing with a professor (or someone who's already been successfully cased, such as the head of faculty), Flash then shows a link to the staff info page.

Next, you move on to the third `case` statement, and if you're dealing with a student (or someone who's already been successfully cased, such as the head of faculty or one of the professors), then that person gets to see a link to the student info page.

Finally, the `default` action generates a message telling the user to go away. Anyone viewing the movie should see this.

Huh? Surely you don't want to tell bona fide members to go away! This highlights the difference between actions in an `else` clause within an `if...else` structure and actions in a `default` clause within a `switch` statement:

- An `else` action will run **only** if all the previous if tests fail.
- A `default` action will **always** run unless you break out before reaching it.

You don't want faculty members to see the default message, so you need to make sure there's a break somewhere before hitting the `default` keyword. No problem:

```
switch (type of user) {
case "head of faculty" :
  show accounts;
case "professor" :
  show staff info;
case "student" :
  show student info;
  break;
default :
  tell user that this site is not for them;
}
```

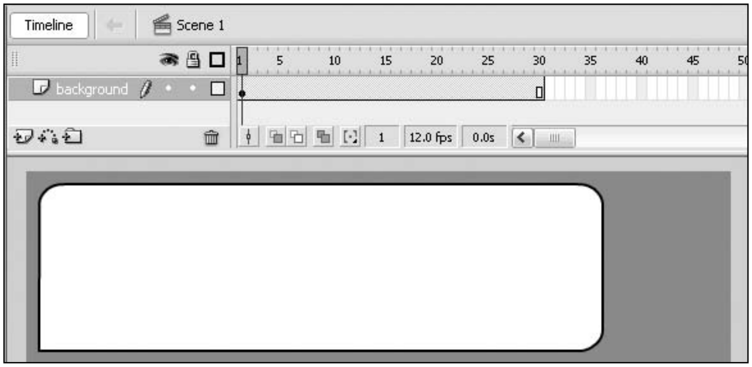
Since all faculty members get to see the student info, they all get to break out before Flash has a chance to tell them to get lost.

The faculty links page

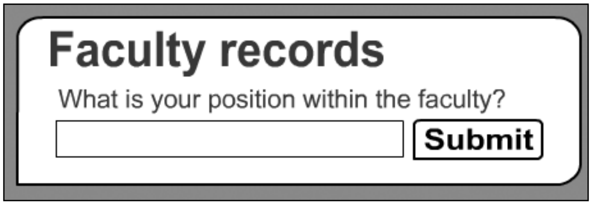
Now that you have a good idea of how your decision-making script should look, let's put it to work.

> *If you want to bypass all the graphics creation in this exercise and jump straight to the code, you can use the file* `faculty_start.fla`*. Using this file, you get to skip straight to step 9 in the following exercise.*

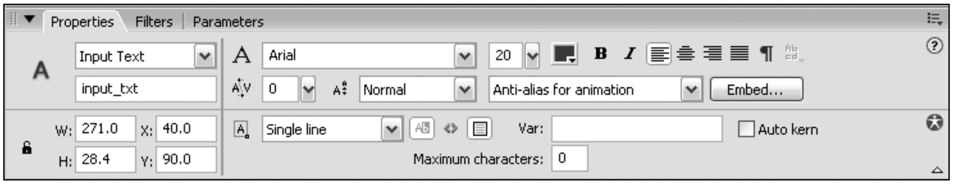
1. Open a new movie and click the stage to select it. Now use the Property inspector to set the background color to a pinkish-purple (we've used #CC66FF). Obviously, this isn't essential, but it does help make things look a little less drab!
2. Now rename Layer 1 as background, and add a white box like the one shown in the following image. Select frame 30 and press F5 to insert a frame there.



3. Now add a new layer called text. This layer will hold all your instructions for the user. On frame 1, you need to see the following:



At the top of the white box, we've created a couple of static text fields. Underneath, there's an input text field, which we've set up like this:



Properties | Filters | Parameters

Input Text

input_txt

Arial | 20 | Normal | 0

Anti-alias for animation | Embed...

W: 271.0 X: 40.0

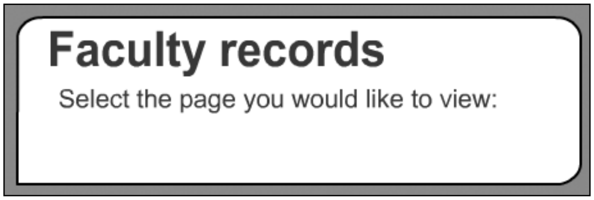
H: 28.4 Y: 90.0

Single line | Var: | □ Auto kern

Maximum characters: 0

Note that we've selected the Show Border Around Text icon and named the instance input_txt. There's also a Submit button (which we've subtly recycled from Chapter 3's calculator movie). The button's instance name is, predictably enough, submit_btn.

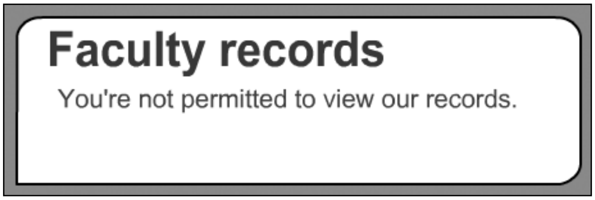
4. Still on layer text, select frame 10 and press F6 to add a keyframe. You can delete the input field and the button from this frame, and change the text in the lower static field so that it looks like this:



Faculty records

Select the page you would like to view:

5. Add another keyframe at frame 20 and change the lower text again, so that it looks like this:

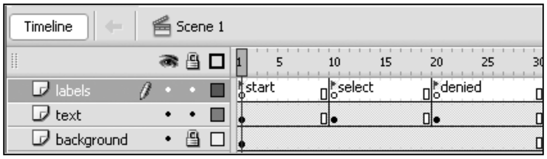


Faculty records

You're not permitted to view our records.

No prizes for guessing what this screen is for!

6. Now it's time to add some frame labels so that the `gotoAndStop()` actions you're going to use for navigation don't get too abstract. Create a new layer called labels and use the Property inspector to create keyframes at frames 1, 10, and 20 and label them start, select, and denied as shown in the following screenshot:



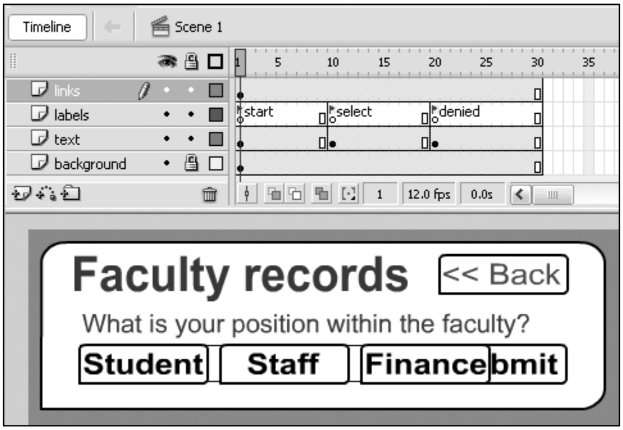
Timeline | Scene 1

labels: start (1), select (10), denied (20)

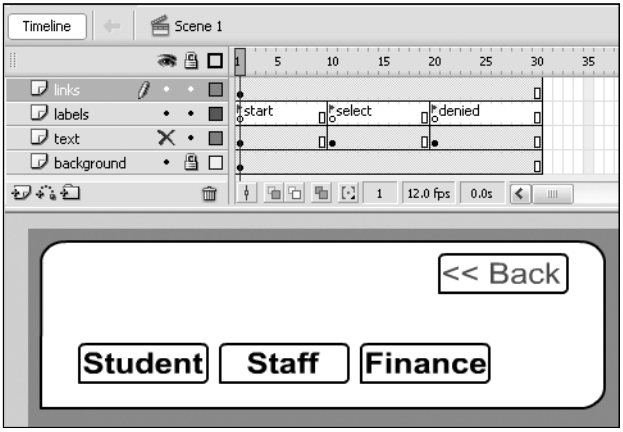
text

background

7. Now for another layer, called links. This is where the actual link buttons are going to sit. There are four of them, each based on its own button symbol in the Library. Again, they're based on the calculator buttons from Chapter 3.



Because these buttons will have to go over the Submit button and text entry field, you may have to hide the text layer while you position your four new buttons like this:



8. Once you've positioned the buttons on the stage, give them the following instance names: student_btn, staff_btn, finance_btn, and back_btn. We'll leave you to work out which one's which!

9. If you run the movie at this point, you'll see some very strange behavior—not at all what you'd hope for. Apart from the fact that the movie is cycling through the entire timeline, the link buttons are visible and active all the way through, and they're covering up the input text field! What on earth have you done?

Don't panic. This is all quite intentional and part of the plan. You're going to make use of a nifty little trick that involves making buttons invisible. Now you see them, now you . . . where'd they go?

10. Create another new layer called actions (if you're using faculty_start.fla, this layer will already be there). Don't worry, we promise this will be the last new layer! Lock the actions layer to prevent placing any graphic elements on it by mistake. Select frame 1, call up the Actions panel, and click the pushpin button, just to make sure you don't accidentally attach any script to the wrong place. Now add the script shown here:

```
// Initialize
var userType:String = "";
// Hide the links
student_btn._visible = false;
staff_btn._visible = false;
finance_btn._visible = false;
back_btn._visible = false;
```

This begins by initializing a variable called userType, which will be used to store the user input. Since the content of an input text box is always text, its type is defined as String. Then all of the buttons are hidden.

Just as you can use _x to control the x-coordinate of an instance on the stage (as you saw in Chapter 1), you can use _visible to control its visibility. By setting _visible to false, you can make an instance invisible.

11. Test run the movie again, and you'll see immediately what a difference this makes! It's still looping, though, so add a stop() action to the end of the script before you go on:

```
back_btn._visible = false;
// Stop the movie
stop();
```

12. Apart from the actual links, your movie offers three ways for a user to interact: via the input text field (where the user states his or her position), the Submit button (which the user clicks to submit the information), and the Back button (which the user clicks to return to the original screen and try again).

The Back button is the easiest, since all it ever needs to do is send you back to the frame labeled start and stop there. Add the following code *above* the existing lines (we'll explain the logic of this position in the section "Organizing your code in the correct order" at the end of this chapter):

```
// Function for 'back' button onRelease event handler
back_btn.onRelease = function() {
  gotoAndStop("start");
};
// Initialize
```

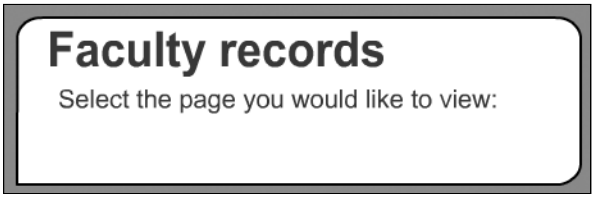
13. The Submit button is more of a challenge. This is what you'll use to trigger your decision-making code (like the _execute button in the earlier examples). The action you want to take for each case is to **show** the relevant button (which will presumably link to some other part of the movie). You can achieve this by resetting the button's visibility to true. Hold your breath and add the following code immediately after the back_btn.onRelease function:

```
// Function for 'submit' button onRelease event handler
submit_btn.onRelease = function() {
  // Actions to perform in any case
  // Store user input, move to "select" frame, and reveal back button
  userType = input_txt.text.toLowerCase();
  gotoAndStop("select");
  back_btn._visible = true;
  // Actions to perform depending on user type
  switch (userType) {
  case "head" :
    finance_btn._visible = true;
  case "professor" :
    staff_btn._visible = true;
  case "student" :
    student_btn._visible = true;
    break;
  default :
    gotoAndStop("denied");
  }
};
```

OK, you can exhale now.

Let's go through this carefully—it's not nearly as hard as it looks!

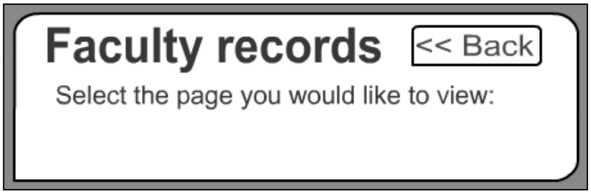
When the user clicks and then releases the Submit button, this triggers the preceding script, which starts by assigning the value of the user input to userType and shuttling the user on to the select frame. At this point, all the link buttons are still invisible, so the stage looks like this:



Faculty records

Select the page you would like to view:

Next, you make the Back button visible. The stage now looks like this:



Faculty records

Select the page you would like to view:

Now it's time to look at the real engine behind this movie. You have a switch action that tests for various different values of userType:

```
// Actions to perform depending on user type
switch (userType) {
case "head" :
  finance_btn._visible = true;
case "professor" :
  staff_btn._visible = true;
case "student" :
  student_btn._visible = true;
  break;
default :
  gotoAndStop("denied");
}
```

If the value inputted is head (or any case-based versions of this, such as Head or HEAD), then the movie shows the Finance button and cascades through all the other cases (showing the Staff button and the Student button) before hitting break.

Similarly, if the value is professor, it shows the Staff button, falls through to show the Student button too, and stops when it hits break.

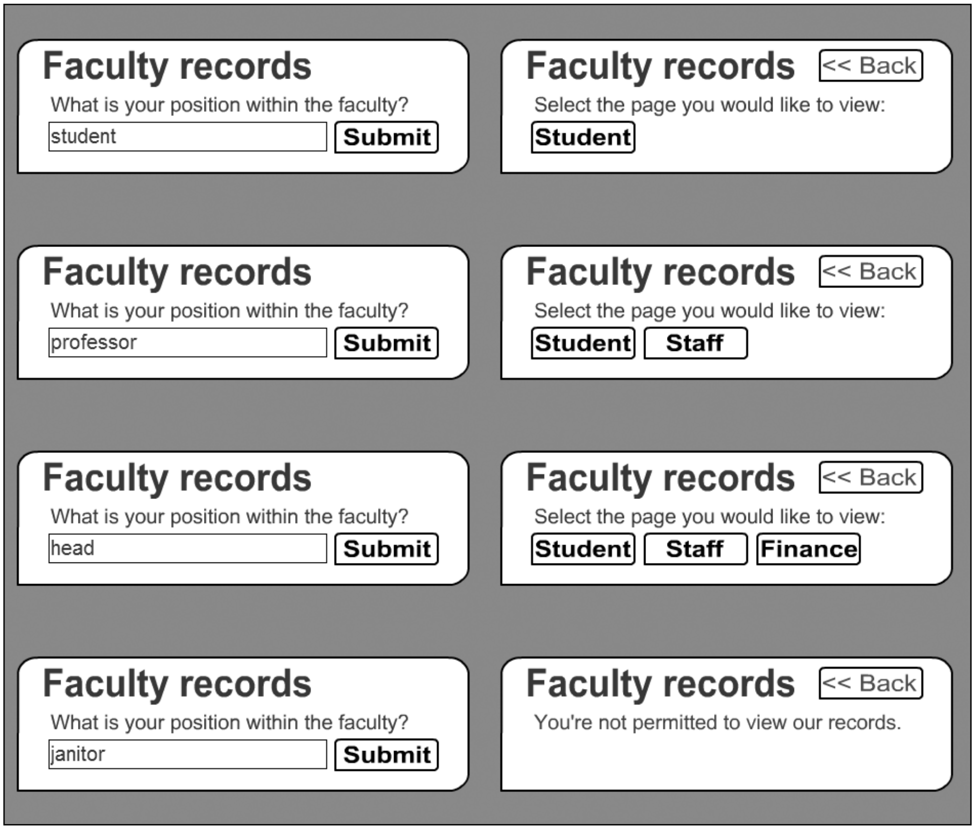
If the value is student, it shows only the Student button before breaking out.

If the value of userType is none of the above, the default clause sends the user hurtling on to the denied frame, where the user is politely informed that he or she doesn't have permission to view faculty records. None of the link buttons is visible, apart from the Back button, which absentminded professors can use to return to the first screen and try to remember their job titles again.

Here's the full script listing for faculty.fla, which you can find in the download for this chapter:

```
// Event handler for 'back' button release
back_btn.onRelease = function() {
  gotoAndStop("start");
};
// Function for 'submit' button onRelease event handler
submit_btn.onRelease = function() {
  // Actions to perform in any case
  // Store user input, move to 'select' frame, and reveal back button
  userType = input_txt.text.toLowerCase();
  gotoAndStop("select");
  back_btn._visible = true;
  // Actions to perform depending on user type
  switch (userType) {
  case "head" :
    finance_btn._visible = true;
  case "professor" :
    staff_btn._visible = true;
  case "student" :
    student_btn._visible = true;
    break;
  default :
    gotoAndStop("denied");
  }
};
// Initialize
var userType:String = "";
// Hide the links
student_btn._visible = false;
staff_btn._visible = false;
finance_btn._visible = false;
back_btn._visible = false;
// Stop the movie
stop();
```

Try the movie out one last time, and you'll find that each different type of user is presented with a different selection of link buttons to choose from, as shown in the following screenshots.



Faculty records
What is your position within the faculty?
student

Faculty records
Select the page you would like to view:

Faculty records
What is your position within the faculty?
professor

Faculty records
Select the page you would like to view:

Faculty records
What is your position within the faculty?
head

Faculty records
Select the page you would like to view:

Faculty records
What is your position within the faculty?
janitor

Faculty records
You're not permitted to view our records.

Organizing your code in the correct order

As you have seen in both exercises in this chapter, the ActionScript code is not always in the order that you might expect. For instance, the `lowerCasePass` and `userType` variables were both defined toward the end of the script *after* the functions in which they were used. Although this seems topsy-turvy, the code is written in the order in which it is actually required by Flash. The standard ordering of code is as follows:

1. Define functions and similar structures. This way, they are ready for action as soon as you click a button or the function is called by another part of the script.
2. Initialize all the timeline variables using `var`. A timeline variable can be accessed anywhere on the same timeline, including functions declared earlier in the script. If you want to restrict the scope of a variable to inside a specific function, however, you should declare it inside the function, again using `var`. We'll return to the question of variable scope in Chapter 6.
3. The main code (everything that's left) comes at the end. The reason for this is that all the functions and variables need to be primed before the main code can run successfully.

Although this order takes a little getting used to, it makes sense when you understand the reasons behind it.

Summary

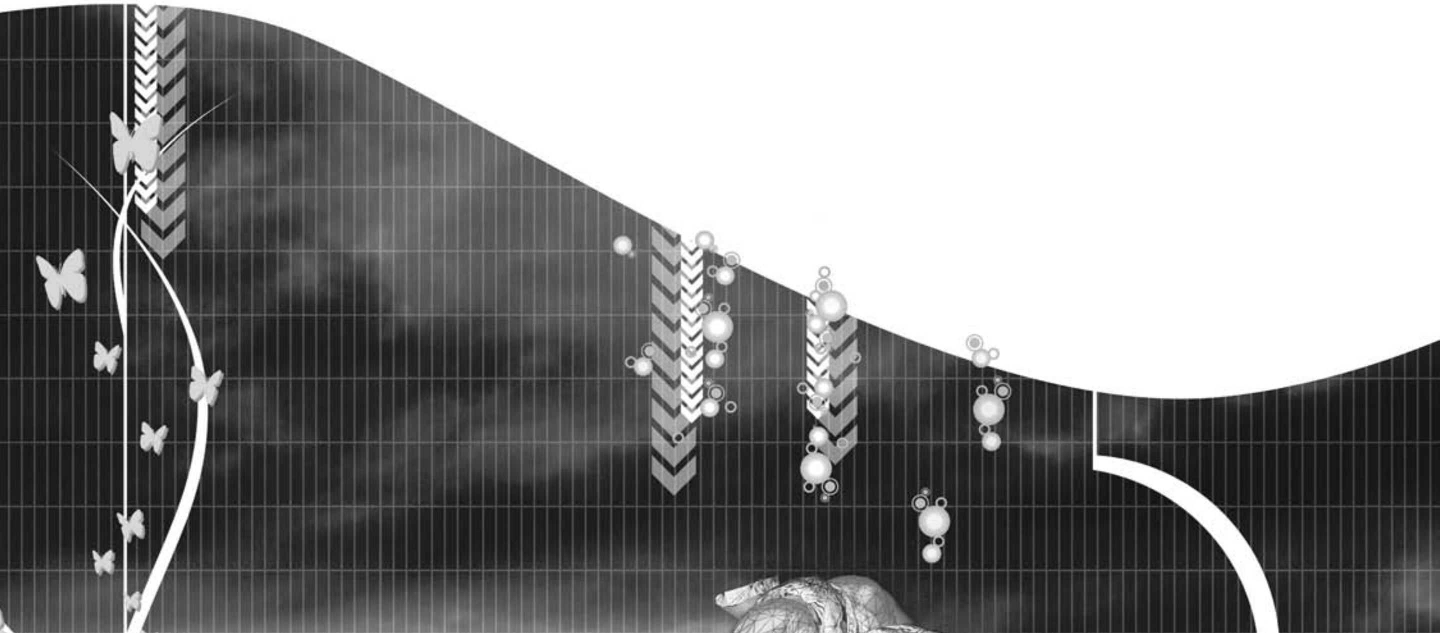
This chapter showed how to make your Flash movies more intelligent by giving them the power to make decisions based on specific circumstances and user input in the following ways:

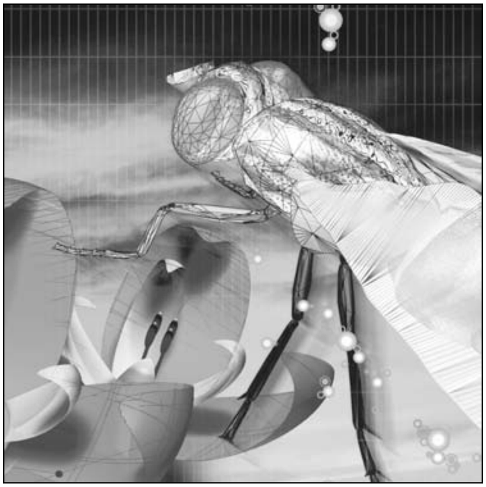
- Using `if` and `else` actions, Booleans, and conditions to enable Flash to make decisions and act on them
- Adding `else if` actions to enable Flash to act on a variety of outcomes
- Using `switch` actions with `break`s to allow one of a number of options to happen
- Using `switch` actions without `break`s to allow several options to happen

You've designed some simple password systems to show this, although we would like to stress—just one last time—that these methods aren't very secure, so don't go using them commercially!

Once you have your head around these basic ActionScript building blocks, you'll be able to go on and create increasingly intelligent Flash movies with the power to implement intricate decision-making code.

In the next chapter, you'll look at how to write your code so that it works harder for you.





Chapter 5

MORE POWER, LESS SCRIPT

What we'll cover in this chapter:

- Using loops to make Flash repeat actions
- Examining several different kinds of loops
- Using loops with arrays to work on lots of information at once
- Building a hangman game with Flash and ActionScript
- Creating the static graphics for the Futuremedia site

It's not exactly headline news, but computers are pretty dumb. Whatever far-fetched dreams we may have once had, the twenty-first century isn't buzzing with the conversation of HAL-9000 supercomputers—well, not yet at least! It's actually chock full of silicon number-crunchers, and (sorry to say) even your shiny new top-of-the-line Pentium can't hold a candle to the raw intelligence of the simplest human being.

When it's a matter of crunching through commands and figures, though, the computer wins easily, both in terms of speed and accuracy. It's specifically designed to work on large, complicated jobs by breaking them down into gazillions of tiny instructions and getting them all done in the blink of an eye, so it's perfectly suited to repetitive jobs.

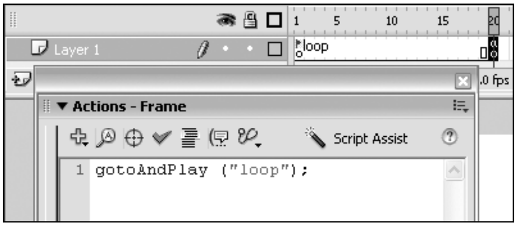
By and large, however, it's human programmers who have to tell a computer what to do and when, and they just don't have time to write out *every one* of those gazillions of instructions—especially when there are reruns of *Star Trek* to be watched on TV! And thus it was that **loops** came into being.

In Flash, a loop is a command (or a collection of commands) that makes your movie do something (or a collection of somethings) over and over again.

Don't worry if this statement doesn't immediately make the whole world seem like a better place to live—it's just a definition of the word "loop." The rest of this chapter will be spent persuading you that loops aren't just useful to Trekkies, but they can save *you* lots of time and effort, and open up some very cool possibilities.

Timeline loops

The simplest way to make a loop in Flash is by using `gotoAndPlay()` to repeat a section of the timeline. For example, to create a timeline loop, you could label frame 1 as loop and attach `gotoAndPlay("loop")` to one of the later frames:



```
gotoAndPlay ("loop");
```

When this timeline plays, it will play normally as far as frame 20, see the `gotoAndPlay()` action, and jump back to frame 1. Then it will play through frames 1 to 20 again before it runs the frame 20 actions again, jumps back to frame 1, and plays from there until it reaches frame 20. And so on, and so on, and so on . . .

So, this will set up a loop between frames 1 and 20, and Flash will quite happily let this repetition go on indefinitely. Of course you might be thinking, "Why bother with a `gotoAndPlay()` action? Flash normally repeats the timeline anyway." Well, there are lots of very subtle differences we could talk about, but there are a couple reasons that make the rest redundant:

- An automatic timeline loop is simple and dumb. It plays the whole timeline. It plays the whole timeline again. It plays the whole timeline yet again. And so on. What's more, any old user can tweak the player settings to turn it off, which is no good if your movie depends on it!
- A `gotoAndPlay()` timeline loop puts you, the author of the movie, in control of the looping. You can specify where the loop starts and where it stops, so it doesn't need to loop the entire timeline. You can even have several loops on the same timeline! You can use it within movie clips too. Even better, none of those pesky loop-stopping users can break it.

Timeline looping can be a very useful technique for getting the most out of what's already there on the stage (e.g., you can keep repeating a tween), but it doesn't give you much of the value-added interactive power that ActionScript promises.

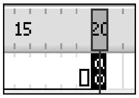
The real subject of this chapter involves another, quite different approach to looping. Instead of repeating a sequence of frames on the timeline, let's see how you can repeat a set of actions.

ActionScript loops

Loops really come into their own when you start using them to repeat scripts that you've already written. Maybe you have a long list of variables that all need to be tweaked in just the same way or lots of movie clip instances on the stage that all need to be shifted 1 pixel to the left. It's easy to write an action to change one of them and not so hard to write a couple more. By the time you've written 428, though, your energy might be starting to flag a little.

If, on the other hand, you write out your action once and stick it inside an ActionScript loop, you can tell the loop to repeat that action as many times as you like. Yes, it's time to capitalize on the fact that your computer will never get bored doing simple, repetitive jobs!

An ActionScript loop is a totally different beast from a timeline loop. A timeline loop involves playing through a sequence of frames, whereas an ActionScript loop exists entirely within a single frame. The ActionScript loop is defined in the lines of script you've attached to that frame, so Flash will stay on the same frame for as long as those lines are looping.



ActionScript has several looping actions specifically designed to help you out with this sort of thing. They all use the sausage notation you saw in the last chapter, so they hopefully won't look too alien to you.

while loops

A **while loop** is probably the simplest of all ActionScript loops to understand, since it's not much different from a simple `if` statement:

```
while (condition) {
  // actions
}
```

In fact, the only thing that *is* different from an if statement is the name of the action. Well, that and the end result, of course! You give the while loop a condition (something that will *always* turn up either true or false) and some actions to perform **while** the condition is true. Unlike in an if statement, Flash won't stop and move on once it has run the actions; it will perform them over and over and over for as long as the condition is true. It will move on to the next line of script (or the next frame) only when the condition turns out to be false.

That means that something like this:

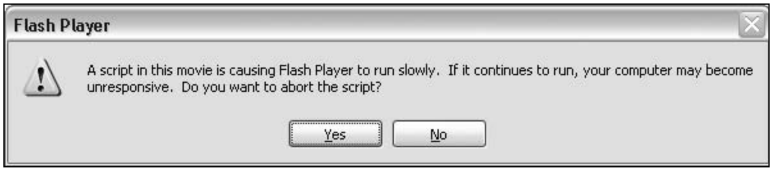
```
while (false) {
  // actions
}
```

won't do a thing. The condition is always false, so any actions you put in the body will never run. On the other hand, something like this:

```
while (true) {
  // actions
}
```

will kill your movie stone dead! The condition is always true, so the loop goes on forever.

You can try making this happen if you like, but be warned: your computer may lock up for a while when Flash tries to compile the SWF. When Flash spots your mistake, it will show you the following message:


Flash Player

A script in this movie is causing Flash Player to run slowly. If it continues to run, your computer may become unresponsive. Do you want to abort the script?

If you do ever see this dialog box, we strongly suggest clicking Yes *every time*. Although Flash can detect these infinite loops, they can still **crash your computer** if you click No and let them run.

Any ActionScript loop that repeats indefinitely can be described as an ***infinite loop****. These will normally only ever happen if something has gone wrong.*

Useful things to do with while loops

So how can you do something useful with a while loop? Simple: Base the condition on a variable, and put actions in the body of the loop that can change that variable. Here's an example:

```
var myCounter:Number = 0;
while (myCounter < 10) {
  myCounter++;
}
```

You set up a variable called myCounter, giving it a value of 0. When you start the while loop, the condition you give it basically says "The value of variable myCounter is less than 10." Well, at this stage, that's most certainly true, so you run the actions inside the body of the loop.

There's only one action in the loop body, and while it may look a little odd, it's really nothing special. myCounter++ is just a shorthand way of saying "Add 1 to the number stored in myCounter." So that's what Flash does, just before it reaches the end of the loop.

Since you're still working through a while action, you loop back to the start. myCounter now has a value of 1, so myCounter < 10 is still true. You therefore run myCounter++ and loop back to the start again. And again. And again. This keeps happening until the value of myCounter reaches 9. At that point, you add 1 to the value, loop back to the start, and test whether the new value is less than 10. It's not! Now the condition is false, so Flash can stop looping and move on.

Of course, this just gives you the bare bones of a useful loop—it's not terribly exciting if it just adds lots of 1s to a boring old number variable. Let's flesh things out so that this simple while loop actually does something practical in a movie.

Making actions run several times in a row

This quick example demonstrates a few of the things it's possible to do with a simple while loop.

1. Create a new Flash document and select frame 1 on the main timeline. Open the Actions panel and click the pushpin button to pin it to the current frame. Now add the following script:

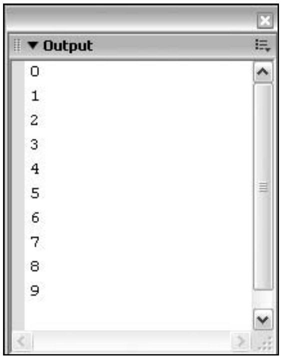
```
var myCounter:Number = 0;
while (myCounter < 10) {
  // actions to loop go here
  myCounter++;
}
```

   This is your basic loop, which will run a complete ten cycles. At the moment though, there is no proof that anything is actually happening.

2. The simplest way to prove that the loop is running is to trace out the value of myCounter for each loop by adding the new line as shown:

```
var myCounter:Number = 0;
while (myCounter < 10) {
  // actions to loop go here
  trace(myCounter);
  myCounter++;
}
```

   Run the movie, and sure enough this is what you get in the Output window:



```
0
1
2
3
4
5
6
7
8
9
```

   This reminds you that myCounter takes values of 0, 1, 2, and so on up to 9, before the loop stops (when it hits 10). It's important to remember that you never actually *see* the value of myCounter traced as 10 because the loop stops as soon as it gets to that value.

This business of counting from 0 to 9 instead of counting from 1 to 10 should remind you of something you saw back in Chapter 3: arrays.

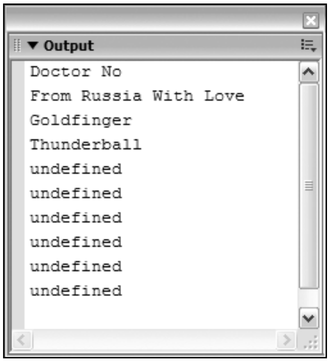
As you may recall, a single array can hold lots of different variables—the **elements** of that array. You can tell Flash which element you want to work with by specifying a particular offset number (or **index**). If you use a variable to specify the offset, the element you deal with depends on the variable's value. Then you can pick and choose from the array's elements by changing the value of that offset variable. You still with us?

Anyway, the upshot of this is that you can write generalized code for manipulating array element i (where i is a variable that contains a number value) and specify the value of i quite separately. If you happen to be looping through all possible values of i, this approach can be *very* powerful!

3. So let's try using myCounter as the offset value used to specify one particular element in an array. Add the following code as shown in bold:

```
  var bondMovies:Array = new Array ("Doctor No","From Russia ➥
   With Love", "Goldfinger", "Thunderball");
  var myCounter:Number = 0;
  while (myCounter < 10) {
    // actions to loop go here
    trace(bondMovies[myCounter]);
    myCounter++;
}
```

Now you get something more like this:

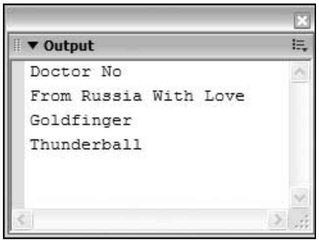


```
Doctor No
From Russia With Love
Goldfinger
Thunderball
undefined
undefined
undefined
undefined
undefined
undefined
```

Instead of just showing the counter value, you're using the counter value to show the first ten values from the bondMovies array. The only problem is that there are just four entries.

4. There's no point in looping ten times if there are only four entries in the array! In fact, you need your loop to loop only for as long as the array element you're looking at has a well-defined value. In other words, you should be looping "while bondMovies[myCounter] is **not undefined**." Sounds like you need to change your loop condition:

```
var bondMovies:Array = new Array("Doctor No", "From Russia ➥
  With Love", "Goldfinger", "Thunderball");
var myCounter:Number = 0;
while (bondMovies[myCounter] != undefined) {
  // actions to loop go here
  trace(bondMovies[myCounter]);
  myCounter++;
}
```



```
Doctor No
From Russia With Love
Goldfinger
Thunderball
```

Run the movie once again, and you'll see that this has done the trick.

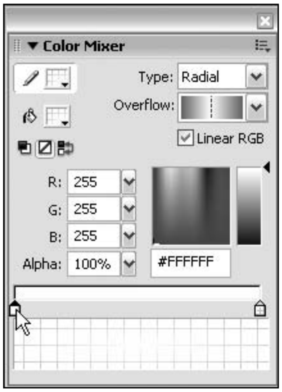
As soon as myCounter reaches 4, the while loop spots that bondMovies[myCounter] isn't defined, so the loop cuts out before it gets a chance to trace the word "undefined."

OK, this may seem like pretty cool stuff if you happen to enjoy making lists of movie titles. But what about doing some real Flash stuff?

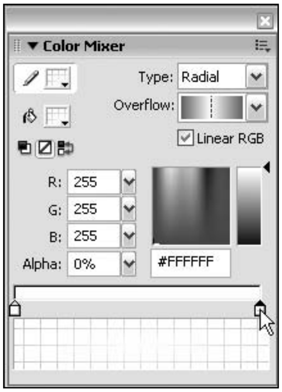
Using loops to control what's on the stage

This time, you're going to use a similar loop to put a whole set of movie clips on the stage. Along the way, we'll introduce some useful tricks that you haven't seen before for controlling movie clips.

1. Create another new Flash document, and give it a black background. Then rename Layer 1 as particles and create a new layer above it called actions. As usual, lock the actions layer.
2. Select the particles layer and use Insert ➤ New Symbol (or press Ctrl+F8/Cmd+F8) to create a new movie clip called particle. Now use the Color Mixer panel to set up a radial fill style that fades from opaque white at the center to transparent white at the perimeter:

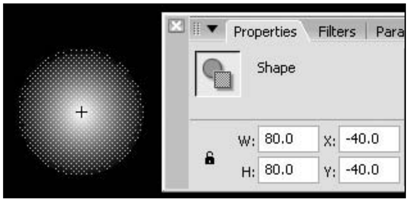


Type: Radial
Overflow:
☑ Linear RGB
R: 255
G: 255
B: 255
Alpha: 100%
#FFFFFF



Type: Radial
Overflow:
☑ Linear RGB
R: 255
G: 255
B: 255
Alpha: 0%
#FFFFFF

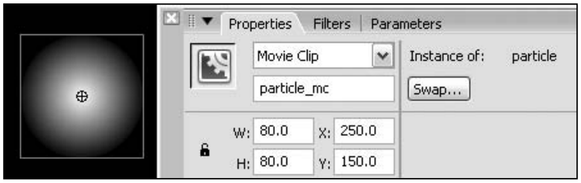
3. Add a circle with no stroke to the particle movie clip. Use the Property inspector to give it a diameter of 80 pixels (W: 80, H: 80) and place it smack in the center of the movie clip registration point (X: –40, Y: –40):



Shape
W: 80.0
H: 80.0
X: -40.0
Y: -40.0

Now click the Scene 1 link to head back to the main timeline.

4. Press Ctrl+L/Cmd+L to call up the Library panel if it's not already visible, and drag the particle item onto the stage to create an instance of the movie clip. Select the instance and use the Property inspector to place it at X: 250, Y: 150. While you're at it, don't forget to give it an instance name: particle_mc.



Movie Clip

particle_mc

Instance of: particle

W: 80.0 X: 250.0

H: 80.0 Y: 150.0

> *Remember, you always put* _mc *at the end of a movie clip instance name to remind you (and, more important, Flash) what sort of instance you're dealing with when you refer to it in the script.*

5. Now select frame 1 of the actions layer, open the Actions panel, and click the pushpin button to pin it to the current frame. Now add the following script:

```
var i:Number = 0;
while (i < 10) {
  // actions to loop go here
  i++;
}
```

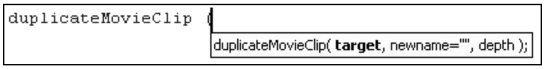
This is just the same loop as you used before, except that you're using a variable called i as your counter instead of myCounter.

So what about all that stuff we said in Chapter 3 about naming your variables sensibly? OK, this is a bit of an exception. When you're writing actions inside loops, you often end up using the counter variable a lot, for array offsets and other such things. If you call it something like myCounter, these actions can get very long and complex-looking. It's therefore quite common to use a variable called i, j, or k instead. We won't bore you with an explanation of *why* these particular names are used; suffice it to say they hark back to the days of mainframe computers that had just 16KB of memory and needed to use short variable names to save memory. We promise it won't take you long to get used to this little quirk.

6. Now you're going to add an action into your loop that will duplicate the particle movie clip:

```
var i:Number = 0;
while (i < 10) {
  // actions to loop go here
  duplicateMovieClip(particle_mc, "particle"+i+"_mc", i);
  i++;
};
```

This is something totally new—and very useful! It doesn't take a genius to work out *what* an action called duplicateMovieClip does, but *how it works* is another matter.



```
duplicateMovieClip (
duplicateMovieClip( target, newname="", depth );
```

This action takes a grand total of three arguments. The first is the instance name of the movie clip you want to duplicate—in this case, it's particle_mc—which doesn't need any quotes around it. The second argument is where you specify what the duplicate should be called. Each instance should have a unique name, so you build this up from two string literals and the counter variable. You're naming each duplicate instance as

```
"particle"+i+"_mc"
```

So when i has a value of 1, this works out as particle1_mc. Likewise, when i has a value of 2, it works out as particle2_mc. And so on up to 9. The third argument tells Flash which **depth level** to place the duplicate on. The only thing you really need to know about this (for now at least) is that you can have only one movie clip on each depth level. If you put two on the same level, the second will *replace* the first. You therefore need to change this argument for each new duplicate, so you use the counter variable i again.

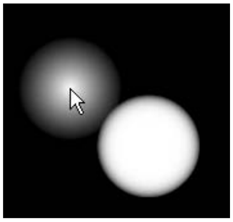
Now test your movie. The end result is that you wind up with ten duplicates of the particle_mc movie clip instance called particle0_mc, particle1_mc, particle2_mc, and so on. All this from one action inside a loop! Let's do something with them.

> *The reason the first argument (the instance name* particle_mc*) didn't require quotes is that you don't mean the literal string* 'particle_mc'*, rather you mean the movie clip whose name is* particle_mc*. The former is a literal, and the latter is a reference. Think of a reference as the same thing as a variable name—it doesn't need quotes.*

7. Now that you have a grand total of 11 movie clip instances on the stage, you can start dragging them around in a variety of weird and wonderful ways. Say you recycle the mouse-following event handler you brewed up at the end of Chapter 1. Add the following bold section where shown:

```
particle_mc.onEnterFrame = function() {
  // move particle to mouse position
  this._x = _xmouse;
  this._y = _ymouse;
};
var i:Number = 0;
while (i < 10) {
  // actions to loop go here
  duplicateMovieClip(particle_mc, "particle"+i+"_mc", i);
  i++;
}
```

As usual, event handlers go before the main code.



Now run the movie, wiggle your mouse around, and watch that particle go!

8. Ahem. Well, you now have *one* particle following the mouse around—what about the other ten? Of course, you don't want to write separate actions to control every single one. That would be rather silly now that you have the power of loops to fall back on!

```
particle_mc.onEnterFrame = function() {
  j = 0;
  while (j < 10) {
    // move particle to mouse position
    this._x = _xmouse;
    this._y = _ymouse;
    j++;
  }
};
var i:Number = 0;
var j:Number = 0;
while (i < 10) {
  // actions to loop go here
  duplicateMovieClip(particle_mc, "particle"+i+"_mc", i);
  i++;
}
```

The only problem with this is that you're now updating the position of `particle_mc` ten times over, when you actually want to reposition a *different* instance each time.

9. Ideally, you want to build another string like `"particle"+j+"_mc"` and use *that* to specify which instance to reposition. Unfortunately, you can't just use this:

```
    "particle"+j+"_mc"._x = _xmouse;
```

because Flash will think that you're trying to assign the value of `_xmouse` to something that isn't a variable or a property. It's not smart enough to spot that you're actually talking about the `_x` property of an instance called `particle0_mc` or whatever.

You need to give Flash a helping hand, and this leads straight into the second crafty trick for the day. Instead of writing this:

```
  "particle"+j+"_mc"._x = _xmouse;
```

you can write this:

```
  _root["particle"+j+"_mc"]._x = _xmouse;
```

The `_root` bit at the beginning tells Flash to look in the main (or **root**) timeline, and the stuff inside the square brackets [] tells Flash "The thing inside the square brackets [] isn't a literal but a *reference* (or, in plain English, a variable name or instance name)." It's as if `_root` is an array whose elements hold things like movie clips and use names instead of numbers. Once you've pulled out the element you want, you can use it as normal. Change the event handler as follows:

```
particle_mc.onEnterFrame = function() {
  j = 0;
  while (j < 10) {
    // move particle to mouse position
    _root["particle"+j+"_mc"]._x = _xmouse;
    _root["particle"+j+"_mc"]._y = _ymouse;
    j++;
  }
};
```

Now you'll find that all the duplicates follow your mouse pointer around, while the original sits still in the middle of the stage. Progress to be sure—but you're not done yet!

10. To finish things off, let's stagger the duplicate particles so that they form a blobby line between the original and the current mouse position. The original particle is centered on X:290, Y:190, so you start by working out the x and y differences between this point and the position of the mouse:

```
particle_mc.onEnterFrame = function() {
  // work out distances from center to mouse
  xDist = _xmouse - 290;
  yDist = _ymouse - 190;
```

The values of 290 and 190 for the x and y values are so because of the movie clip's central registration point. 40 pixels (half the height and width of the movie clip) has been added to the normal x and y values (250 and 150, respectively) to get these values.

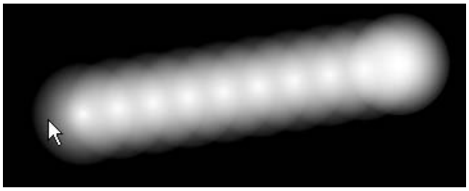
Now you're ready to move on to your loop, where you'll move each particle to X:290, Y:190 plus j tenths of the distance to the mouse. You also need to (rather, you should) initialize your new variables xDist and yDist:

```
particle_mc.onEnterFrame = function() {
  j = 0;
  while (j < 10) {
    // move particle to center plus j tenths of the distance
    _root["particle"+j+"_mc"]._x = 290 + (xDist * j/10);
    _root["particle"+j+"_mc"]._y = 190 + (yDist * j/10);
    j++;
  }
};
var i:Number = 0;
var j:Number = 0;
var xDist:Number = 0;
var yDist:Number = 0;
```

```
while (i < 10) {
  // actions to loop go here
  duplicateMovieClip(particle_mc, "particle"+i+"_mc", i);
  i++;
}
```

This means that particle0_mc will be at the same place as particle_mc. particle1_mc will be one-tenth of the way to the mouse pointer, particle2_mc will be two-tenths of the way there, and so on, until you reach particle9_mc at nine-tenths of the way to the pointer.

Test the movie again, and you'll see some fairly impressive results, even if you did need to do a little math to get them.



It's still not *quite* what you set out to achieve, though. There's a big fat blob at one end (where particle0_mc sits on top of the original) and a conspicuous gap at the pointer itself, where you have no particle at all. This all boils down to that old "counting from 0" business mentioned earlier. This is one time when it would be much more useful if you could start counting from 1 (placing the first duplicate at one-tenth of the distance) and finish at 10 (placing the last duplicate at ten-tenths of the distance; smack-dab on top of the pointer).

11. In fact, it turns out that nothing could be simpler. Both of your loops currently use i++ (or j++) to bump up the loop counter at the end of each cycle. So what about changing the counter at the *start* of each loop instead? You'll need to change both loops, so here's the full script:

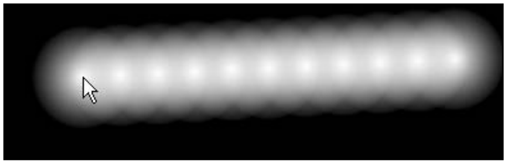
```
particle_mc.onEnterFrame = function() {
  // work out distances from center to mouse
  xDist = _xmouse-290;
  yDist = _ymouse-190;
  j = 0;
  while (j < 10) {
    // move particle to center plus j tenths of the distance
    j++;
    _root["particle"+j+"_mc"]._x = 290 +(xDist*j/10);
    _root["particle"+j+"_mc"]._y = 190 +(yDist*j/10);
  }
};
var i:Number = 0;
var j:Number = 0;
var xDist:Number = 0;
var yDist:Number = 0;
```

```
while (i < 10) {
  // actions to loop go here
  i++;
  duplicateMovieClip(particle_mc,"particle"+i+"_mc", i);
}
```

Now you start each loop with a counter value of 0. The first thing you do is bump it up to 1, use that value to do stuff with particle1_mc, and then loop back. The counter is less than 10, so you run the loop actions again: bump the counter up to 2, use it to do stuff with particle2_mc, and then loop back again.

You keep doing that until the counter is 9. That's still less than 10, so you run the loop actions, which bump the counter up to 10; use it to do stuff with particle10_mc; and loop back again. Only now does the while() action wise up to the fact that its counter<10 condition *isn't actually true anymore*.

Run the movie one last time, and you'll see a beautiful, homogenous line of particles stretching out from the middle of the stage to wherever your mouse pointer happens to be. Presto!



One final thing worth mentioning is that when you need to create lots of variables, Flash allows you to create more than one per var. You could have done this instead of your four separate var lines:

```
var i:Number = 0, j:Number = 0, xDist:Number = 0, yDist:Number = 0;
```

Because you also define the values of all your variables in the code after initialization, there's no need to set them all to 0, so this is also permissible (and used often by the . . . um . . . lazier programmers):

```
var i:Number, j:Number, xDist:Number, yDist:Number;
```

Be careful when you use this, though, because not defining variables as soon as you create them isn't good programming practice—you might start seeing your old friend undefined cropping up in your code unexpectedly. Quick shortcuts are fine, but as in real life, an easy life is hard work. There are usually no shortcuts, only unfinished jobs that come back to haunt you later!

OK, now that you've seen a little of what you can do with a while loop or two, let's take a quick look at the other sorts of loops available in ActionScript. "Other sorts of loops?" we hear you cry. "Surely there can't be *more* ways to send scripts running around in circles?" Well, no . . . and yes.

Most of the loops you've looked at so far use a pretty standard system: set up a counter variable, add 1 to the counter variable's value every cycle, and test each time whether the variable's value has gone above a certain value. This approach is *so* ubiquitous that there's a special action devoted to it.

for loops

Whenever you build a loop that uses some kind of counter variable to count how many cycles you've gone through, it's worth considering the humble `for` loop as an option. It doesn't actually add anything new to the possibilities offered by `while()`, but it lets you set things out in a way that brings all the counter-specific actions together on one line, neatens things up, and helps you keep things nice and clear.

Let's consider a loop you've already looked at:

```
var myCounter:Number = 0;
while (myCounter < 10) {
  trace(myCounter);
  myCounter++;
}
```

There are four important parts to this loop, one on each of the first four lines:

- Initialize the counter variable `myCounter`.
- Specify the loop condition `myCounter < 10`.
- Perform the main loop action `trace(myCounter)`.
- Bump up the value of the counter with `myCounter++`.

If you rewrite this loop as a `for` loop, you can cut it down to two lines:

```
for (var myCounter:Number = 0; myCounter < 10; myCounter++) {
  trace(myCounter);
};
```

You've now defined the whole range of the loop within the first line, so there's no need to set up a counter before the loop and no need to change its value as one of the looped actions. ActionScript takes care of all that for you.

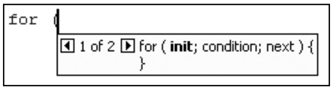
You're probably not convinced yet. These `for` loops certainly don't lend themselves to useful explanations like "While the counter is less than 10, perform the following actions," and they *do* take a bit of getting used to.

The crucial thing to register is the fact that every factor that could possibly influence how many cycles this loop goes through is ***specified in the*** `for()` ***action itself****. Since these factors are all in one place, you'll find it a lot easier to keep track of what the loop is doing when you come across it buried in a mountain of other code.*

Since you can see how many loops a `for` *loop needs to cycle through, it's less likely to give you a nasty surprise when you test your code. Also, you can easily modify a* `for` *loop so that it cycles only once, which can be tremendously useful while developing your code, since if you can see it working for a single loop, you can be confident that it will run for many cycles. In fact, the* `for` *loop is so useful that in most code it's the only loop structure you'll use.*

Of course, you can still use the counter for all sorts of things other than simply counting how many times you've looped—for example, as the index for an array or variable, or for deriving a sequence of numbers (such as a multiplication table; 5, 10, 15, 20, 25, and so on).

Let's take a closer look at each of the three arguments you stick in the `for()` action:



```
for (
    1 of 2  for ( init; condition; next ) {
                }
```

init

The first argument is where you set up the counter variable. It's normal to set this to 0, though you're free to use any value you like. You can also name the counter as you wish. The total number of loops must always turn out to be a numeric variable, so valid start counter values are as follows:

- `counter = 0`: The counter starts at 0.
- `i = startValue`: The counter `i` starts at the value of numeric variable `startValue`.

condition

This is just like the condition in an `if` statement or a `while` loop: as long as the expression evaluates to true, the loop will continue cycling. Inside a `for` loop, though, the condition should always refer explicitly to the counter, normally in a form like "The counter is less than the stop value." Valid examples are as follows:

- `i < 10`: Loop as long as `i` is less than 10.
- `counter >= 200`: Loop as long as `counter` is greater than or equal to 200.
- `x < xMax`: Loop as long as `x` is less than `xMax`.

In the last case, it's important that you **don't let your loop actions change the value of** `xMax`—well, not unless you know what you're doing (in which case you'll probably be using another kind of loop anyway!). Flash won't stop you from doing this, but it will make your code very unpredictable, and it may even result in an infinite loop (yes, the type that can crash your computer!).

One condition that beginners often get stuck with is this:

```
counter == 10
```

Although this is OK as a condition, it's probably not what you want for a `for` loop—unless `counter` equals 10 on the first loop, it will stop the loop immediately and not do anything at all.

next

This is where you specify what should happen to the counter after every loop. Possible arguments you can add here are as follows:

- `i++`: Add 1 to `i`.
- `counter--`: Subtract 1 from `counter`.
- `i = i+2`: Add 2 to `i`.

You've already encountered the ++ trick for adding 1 to the value of a variable. Its partner in crime is --, which takes 1 *off* the value. Anything else you want to do to the counter needs to be written out in full.

Some useful examples of for loops

If you want to see the following for loops in action, simply use the Actions panel to attach them to frame 1 of a new movie and press Ctrl+Enter/Cmd+Enter to give the movie a test run. The Output window will then show you the counter value for each cycle (or iteration) of the loop.

Simple loop

This loop will count up from 0 to 12:

```
for (var counter:Number=0; counter <= 12; counter++) {
  trace (counter);
}
```

Reverse loop

This loop will count down from 12 to 0:

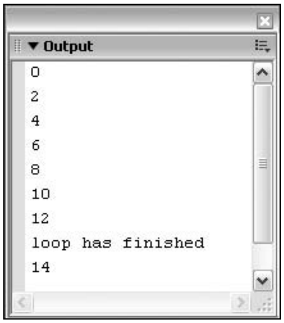
```
for (var counter:Number=12; counter >=0 ; counter--) {
  trace (counter);
}
```

Two at a time

This loop will count up from 0 to 12 in steps of two and show the value of counter once the loop has terminated:

```
for (var counter:Number=0; counter<=12; counter=counter+2) {
  trace(counter);
}
trace("loop has finished");
trace(counter);
```

This demonstrates that the value of the counter just after the loop completes is the first value that doesn't satisfy your condition. In this case, that's two more than the maximum value of 12, namely 14.



```
0
2
4
6
8
10
12
loop has finished
14
```

Looping through elements in an array

As we've now established, loops are pretty darn useful for making a lot of actions run in a very short time. In particular, for loops give you a tidy way to cycle through lots of values with just a couple of lines of code. One particularly useful thing you can do with loops is use them to work with arrays.

Say you've written a set of actions that all act on element i of an array. By looping through all available values of i, you can quickly apply the actions to every element in the array. So a small piece of code can be made to work on large amounts of data.

You already saw a simple example of how you can use a while loop to look at all the elements in an array, and you even persuaded it to stop when the array ran out of defined elements. In fact, the combination of loops and arrays is *so* important that you're going to return to it now and look at a few more things they enable you to do.

Applying an operation to all the elements in an array

Consider an array of 200 values between 1 and 20 that you've stored in an array called rawData_array. You'd like to create a corresponding list of percentage values in a second array called percent_array. The nonloopy way to do this (or should that be "the *totally* loopy way"?) would be to write out 200 lines of script like this:

```
percent_array[0] = rawData_array[0]*5;
percent_array[1] = rawData_array[1]*5;
percent_array[2] = rawData_array[2]*5;
percent_array[3] = rawData_array[3]*5;
...
percent_array[198] = rawData_array[198]*5;
percent_array[199] = rawData_array[199]*5;
```

The sensible way to do it would be to use a loop, and since you know how many elements there are, it makes sense to use a for loop. First, you need to work out a generalized action for the "convert to percent" process.

That's not too hard: percent just means "out of 100," and 1 in 20 is just the same as 5 in 100. So you just need to multiply all your numbers by 5. Assuming you're using a counter variable called i, you can write this:

```
percent_array[i] = rawData_array[i]*5;
```

Now you loop through all values of i from 0 to 199, applying your line of code each time until you've converted the entire array:

```
var i:Number = 0;
var percent_array:Array = new Array();
var rawData_array:Array = new Array();
// code to set values of rawData_array goes here
// ...
// ...
for (var i:Number = 0; i < 200; i++) {
  percent_array[i] = rawData_array[i]*5;
}
```

What if you don't know how many elements there are in the array? You've already seen one way to deal with this issue using a while loop, which you could translate into for-style notation:

```
for (var i:Number = 0; rawData_array[i] != undefined; i++) {
  percent_array[i] = rawData_array[i]*5;
}
```

Hmm. Well, Flash won't actually *stop* you from doing this, but it goes against everything you've just learned about for loops. Sure, the loop condition still depends on i, but only loosely. It depends a lot more on the contents of the rawData_array array, and it's not at all obvious how many times you expect to loop. This is one situation in which things are best left to a while loop.

On the other hand, if you had some way to ask the array directly how many elements it had, then you could still get away with a condition like i < lengthOfArray. Of course, there's just such a way: every array has a property called length, which tells you how many elements it contains. So it looks like your best bet is to use i < rawData_array.length like this:

```
for (var i:Number = 0; i < rawData_array.length; i++) {
  percent_array[i] = rawData_array[i]*5;
}
```

The length *property is common to all arrays and tells you the number of elements an array contains. This technique gives you a very simple way to put limits on all your array loops.*

Searching an array for a specific value

Another thing you can do using a combination of loops and arrays is search through the elements in an array until you find a particular value. Say you have an array of book titles called library_array, and you use a variable called searchString to hold the name of a book you're looking for. There are two ways you could approach this.

You could write this:

```
var searchString:String = "City of Glass";
var i:Number = 0;
while (library_array[i] != searchString && i < library_array.length) {
  i++;
}
if (library_array[i] == searchString) {
  trace("Match found: item number " + i);
} else {
  trace ("No matches found.");
}
```

You don't know in advance how many loops you're going to need before you find a match, so you set up a while loop. You then keep adding 1 to the value of i until library_array[i] matches your search string or you run out of array elements. You then use an if...else to check whether the loop ended because a match had been found (in which case you trace out the relevant element number) or because it got to the end of the array without finding any matches at all (in which case you output the number of matches as zero).

On the other hand, you might want to do it like this:

```
var searchString:String = "City of Glass";
var matchesFound:Number = 0;
for (var i:Number = 0; i < library_array.length; i++) {
  if (library_array[i] == searchString) {
    trace("Match found: item number " + i);
    matchesFound++;
  }
}
trace("Number of matches found: " + matchesFound);
```

This time, you assume that you're going to search the whole library, on the basis that there may be more than one copy of the book you're looking for. You therefore use a `for` loop, which cycles as many times as there are elements in the array (thanks go, once again, to the `length` property). Every time around, the `if` statement will check whether there's a match with the search string. If there is, you trace out details of the match and bump up the number of `matchesFound`. Once the whole array has been searched, you trace out the total `matchesFound` value.

Cross-indexing a pair of arrays

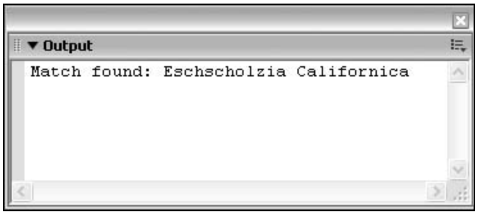
Once you've figured out how to search an array, you're only one step away from cross-indexing with another array. Let's consider two arrays, each containing a list of plant names. Each element in the first array contains the common name of a plant, while the corresponding element in the other array holds the Latin name of the same plant. They might look like this:

```
var commonNames_array:Array = new Array();
commonNames_array[0] = "Foxglove";
commonNames_array[1] = "Palm Tree";
commonNames_array[2] = "Rose";
commonNames_array[3] = "California Poppy";
commonNames_array[4] = "Witch Hazel";
var latinNames_array:Array = new Array();
latinNames_array[0] = "Digitalis";
latinNames_array[1] = "Trachycarpus";
latinNames_array[2] = "Rosa";
latinNames_array[3] = "Eschscholzia Californica";
latinNames_array[4] = "Hammamelis Mollis";
```

What if you know the common name of a particular plant but want to find out what its Latin name is? You might ask the question, for example, "What is the Latin name for the California Poppy?" You just need a loop that will cycle through elements in the `commonNames_array` array until it finds one that contains the string `"California Poppy"`. It could then use the successful index value (in this case 3) to call up the corresponding element from the `latinNames_array` array and find the answer: "Eschscholzia Californica."

Here's how you might do it:

```
var offset:Number = 0;
var search:String = "California Poppy";
while (commonNames_array[offset] != search && offset < commonNames_array.length)
{
  offset++;
}
if (commonNames_array[offset] == search) {
  trace("Match found: " + latinNames_array[offset]);
} else {
  trace ("No matches found.");
}
```



```
Match found: Eschscholzia Californica
```

In this case, you don't necessarily want to search on every element—you'll stop just as soon as you've found a match. Since you can't know in advance how many elements you'll need to search on, a for loop isn't suitable, so you use a while structure. As before, it loops around incrementing your offset variable for as long as the element commonNames_array[offset] doesn't match your search term and offset is less than the array length.

So, when the while loop finishes, it could be for one of two reasons: either you've found a match or you've reached the end of the array and found none. This method—finding an index and applying it to another related array—is known as **cross-indexing**.

Now that you've seen a few theoretical examples of combining loops with arrays, it's time to put your newfound skills to work. You're going to use Flash and ActionScript to create a game of hangman using arrays and loops together with more than a little string variable manipulation. As ever, a nice practical example should help lift the whole thing right off the page!

Hangman

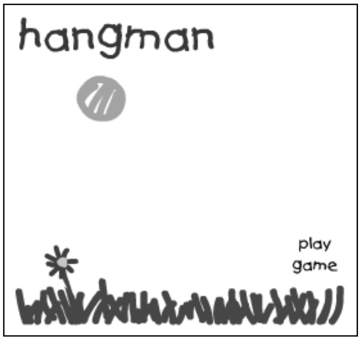
It's an old game that's been around for centuries. Pick a word at random, write it out using a blank in place of each letter, and challenge a friend to guess what the word is by picking letters from the alphabet. If he chooses a letter that's in the word, you show him where it is in the word by filling in the blank. If he picks one that isn't in the word, you add pieces to a drawing of a scaffold. After five or six wrong guesses, the scaffold is complete, and you start adding a stick figure hanging from the gibbet. After about ten wrong guesses, the figure is complete—HANGMAN! The only way for your friend to win is to guess all the letters in the word before the stick figure is hanged.

Hangman is a brutal game when you think about it, and there's no hiding the fact that it's a game about someone getting hanged. All the fluffy bunnies in the world aren't going to soften the blow, but we've decided all the same to get our own version looking like it was drawn and painted by a child.

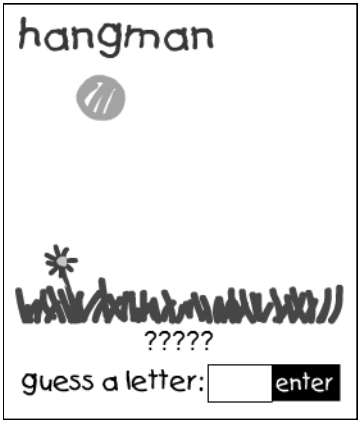
Let's start with a quick walk-through to give you an idea of how the gaming experience should pan out for your players.

1. When you start the movie, you'll see the following scene:

   An empty field on a sunny day, with a title, a solitary flower, and play game button that you can click to start the game.

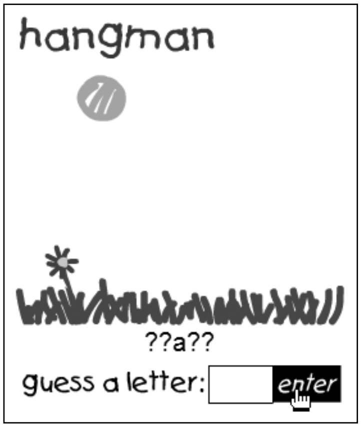


2. Once you click the play game button and the game starts, you're presented with a simple interface: a box for typing in your guesses, an enter button to submit your guesses, and a list of question marks indicating how many letters there are in the word you have to guess:



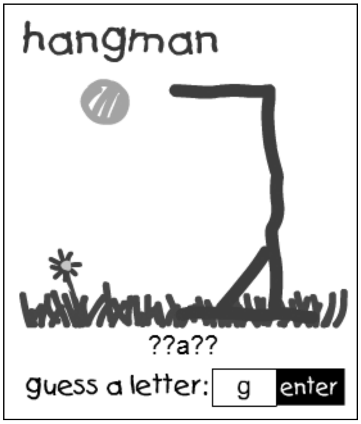
3. The next step is to type in a letter and click the enter button. Let's say the mystery word is "flash" and you've just tried the letter "a." This is what you should see:

   There's one "a" in the word "flash," so Flash replaces the relevant question mark with the letter itself.



4. Say you now try a few more letters, such as "u," "x," "q," and "z." OK, let's just say you're deliberately aiming for incorrect answers here. None of these letters occurs in the word "flash," so they're all bad guesses—now you have to pay the price:

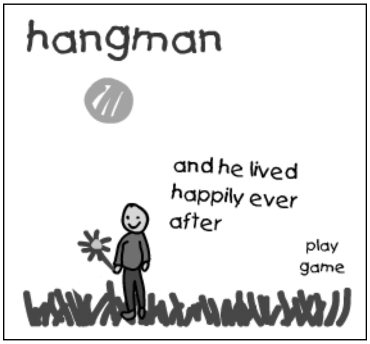
   You've guessed four wrong letters, so four sections of the gallows appear. If you don't start making some decent guesses soon, you'll be hung out to dry! As you can see from the screenshot to the right, you're just about to guess the letter "g." You haven't clicked enter yet, so all is not lost.



*In the advanced version of this game (*hangman_advanced.fla*) the player cannot re-enter a letter that she has already guessed. This is done by preventing the user from re-entering a character into the input text field using the text field* restrict *property. So, if the player tries entering "x" a second time, she'll find that she's not able to enter it.*

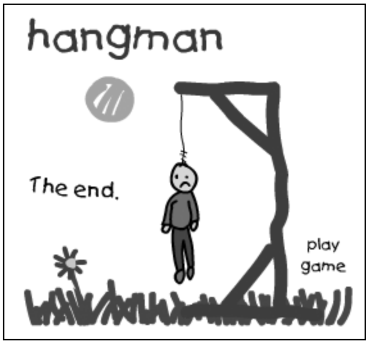
5. Let's assume you now replace the "g" with an "f," click enter, and suddenly realize what the word is. You follow up with "l," "s," and "h," and the word is complete. This is what you should now see:

   The gallows vanishes, and your would-be victim is now free to wander through the sunlit fields with a big red flower in his hand. How sweet.



6. However, there's an alternate reality in which you didn't recant your faith in the letter "f" . . . you left a "g" in the input box and clicked enter! With no clue as to what the word could be, you floundered around with such desperate guesses as "m," "n," "o," "p," and "q." There was no hope:

   Our hero meets a grisly end, and the only solace to be found is in the little button that reappears behind the gallows, hinting that you can actually have another try.



Well, that's the game—and we did warn you things might get a little unpleasant! Now that you have a good idea of how the game "hangs" together, let's start looking at how you can build it in Flash.

> *If you're in a rush and don't want to create these graphics from scratch, you can find them all in this chapter's download, in the Library of a file called* `hangman_start.fla`.
>
> *If you want to concentrate on typing in the scripts only, you can start with* `hangman_startScript.fla`*, which has everything in place except the scripts. Using this file, you can get away with just reading the text up until the section that starts "Adding in the ActionScript."*
>
> *In case you're wondering, the font used throughout the game is called* ***kids****. If you don't have a similar font, just use something else that looks as wacky!*

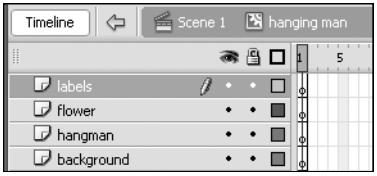
You'll start by breaking down the game into symbols that you need in the Library. We've used a few graphic symbols (called flower, grass, and sun), plus two buttons (enter and play), and we've put all the "hanging" material and interface stuff into a couple of movie clips (called `hanging man` and `interface`).

There's nothing terribly special about the buttons (one's a black box behind a bit of static text that says "enter," and the other's just a bit of text that says "play game." Select them and press F8 to convert them to symbols and you're done), and the graphics should be self-explanatory (the sun is an orangey circle, the title says "hangman," and so on. Select them and press F8 again . . . you get the idea), so we'll skip to how you build the two movie clips.

The hanging man movie clip

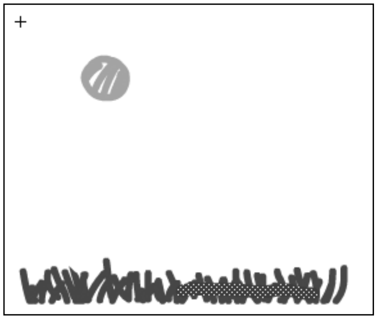
This is where you pull together your scenery and animate the awful process of a hanging.

1. First, you need to create a total of four layers in a new movie clip, `hanging man`, which you should call labels, flower, hangman, and background:



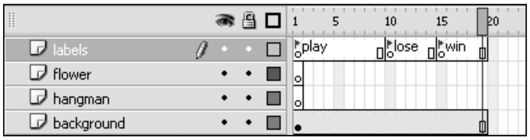
labels
flower
hangman
background

2. Select the background layer, and pull a copy of the sun and grass symbols from the Library onto the stage:

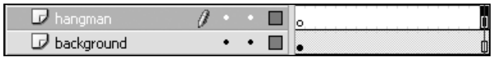


Select frame 19 and press F5 to insert a frame. You want this scenery on display throughout the clip.

3. Now select the labels layer. Press F5 to insert a frame at 19 of the labels layer, and press F6 to add keyframes at 10 and 15 on this layer. Use the Property inspector to label your three keyframes as play, lose, and win, respectively. Your timeline should now look like this:

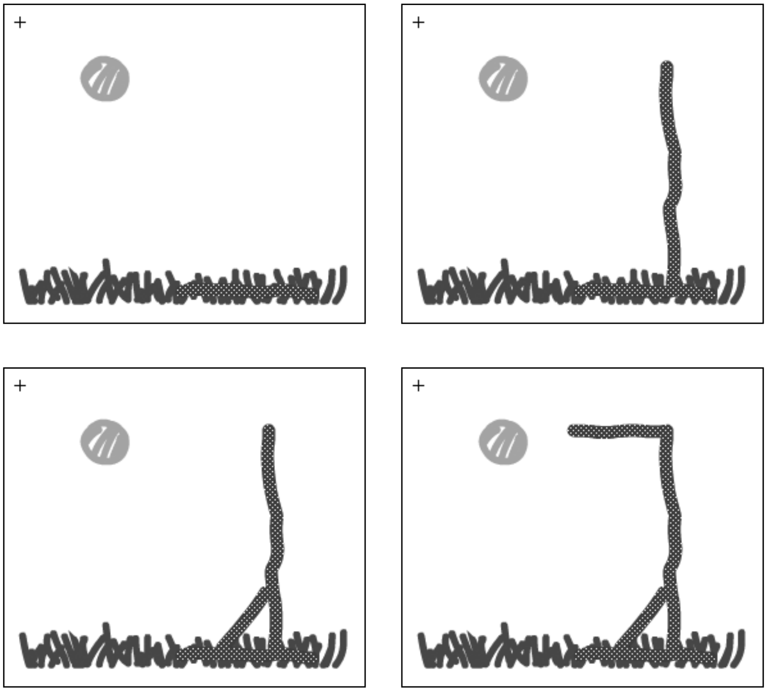


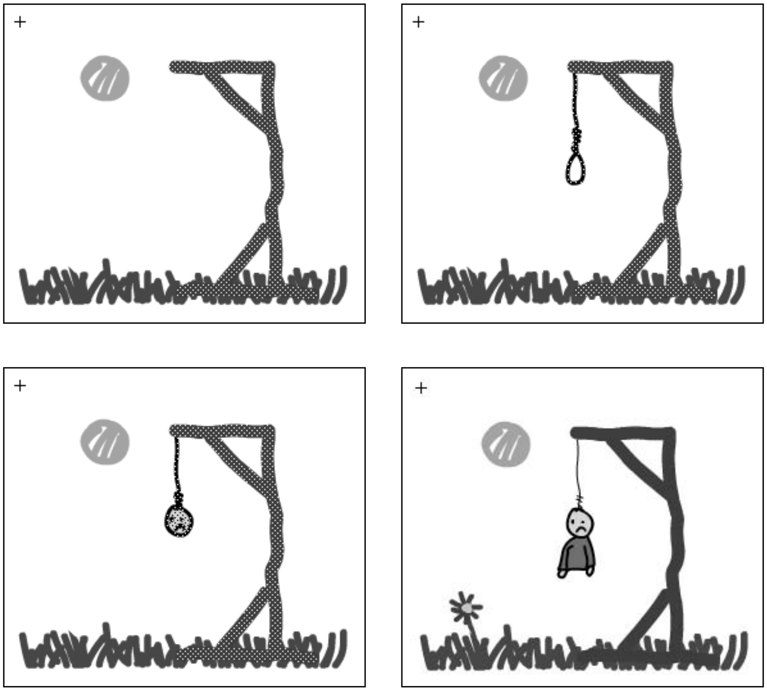
4. Now for the creative part. You're going to add the frame-by-frame animation of the gallows and hanging man. Select frame 19 of the hangman layer and press F5 to insert a frame. Layer hangman should now look like an albino version of the background timeline.



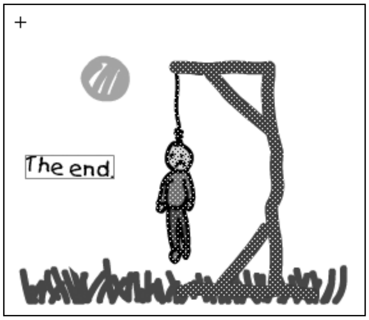
Select frame 2, press F6 to add a keyframe, and draw a horizontal brown line along the top of the grass. Select frame 3, press F6 to add a keyframe, and add a vertical brown line at the right end. Select frame 4, press F6 to add a keyframe, and add a diagonal brown line connecting the two lines you just drew.

Now keep going, building up the gallows a piece at a time and then adding your hapless victim (one piece at a time) until you reach frame 10, which is where folks go when they've lost the game. Each keyframe represents one step closer to doom!





By frame 10, you should have something like this:

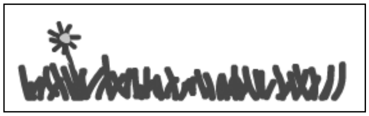


The text the end also appears here (still on the hangman layer), just to emphasize the fact that the player's tale is over.

5. Of course, if the player has won the game, you need to reward his or her efforts and give the player a nice uplifting image. Select frame 15 and press F6 to add a keyframe. You can delete the gallows from this frame and replace it with something happier:

and he lived
happily ever
after

6. Now for a nice finishing touch. Select frame 1 of the flower layer, drag a copy of the flower symbol onto the stage, and nestle it down somewhere safe in the grass:

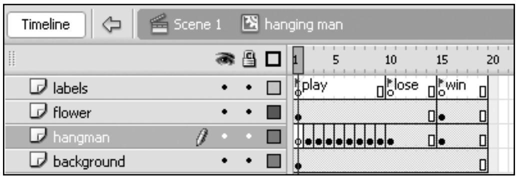


Now add a keyframe at frame 15 of layer flower and drag the flower across so that it looks as if it's being held in the hero's hand:

Select frame 19 and press F5 to insert a frame.

and he lived
happily ever
after

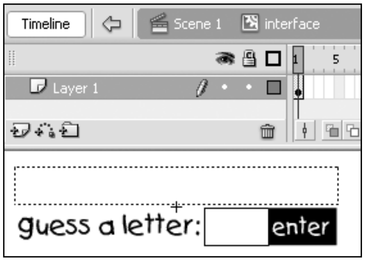
Phew! After all that hard work, your "hanging man" timeline should look like this:



Aren't you glad we didn't show you that before you started? Now that you've dealt with the aesthetics of the game, let's take a look at the user interface.

The interface movie clip

You'll be glad to hear that this movie clip is a great deal simpler than the last one. There's one layer, one keyframe, and no fancy graphics to labor over. In fact, the following image sums it up quite well:



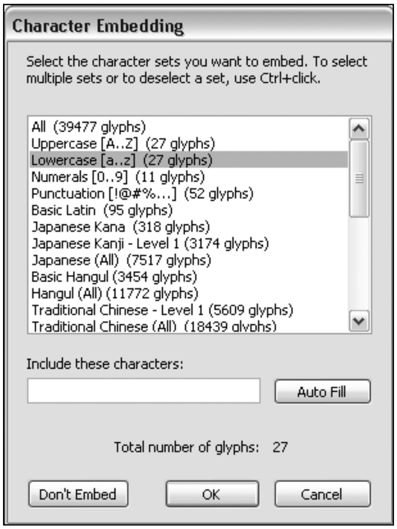
There are just four things on the stage:

- A dynamic text field called `display_txt` at the top
- A static text field on the left that reads guess a letter:
- An input text field called `input_txt`
- An instance of the enter button, `enter_btn`

You want the player to use only lowercase characters, so select the input text field and click the Embed button in the Property inspector. In the Character Embedding dialog box, select Lowercase Letters [a...z] (27 glyphs), and click OK to confirm.

You also want the user to guess a letter at a time, so in the Maximum Characters box on the Property inspector (it's just below the character buttons) enter 1.

Once you've created each of these and set them up on the stage as described, you're all done. Now you're ready to take a look at the main timeline, arrange your movie clips, and wire them all up with a little ActionScript.



Character Embedding

Select the character sets you want to embed. To select multiple sets or to deselect a set, use Ctrl+click.

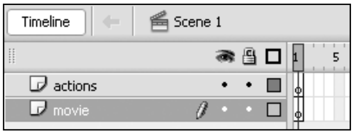
All (39477 glyphs)
Uppercase [A..Z] (27 glyphs)
Lowercase [a..z] (27 glyphs)
Numerals [0..9] (11 glyphs)
Punctuation [!@#%...] (52 glyphs)
Basic Latin (95 glyphs)
Japanese Kana (318 glyphs)
Japanese Kanji - Level 1 (3174 glyphs)
Japanese (All) (7517 glyphs)
Basic Hangul (3454 glyphs)
Hangul (All) (11772 glyphs)
Traditional Chinese - Level 1 (5609 glyphs)
Traditional Chinese (All) (18439 glyphs)

Include these characters:

Total number of glyphs: 27

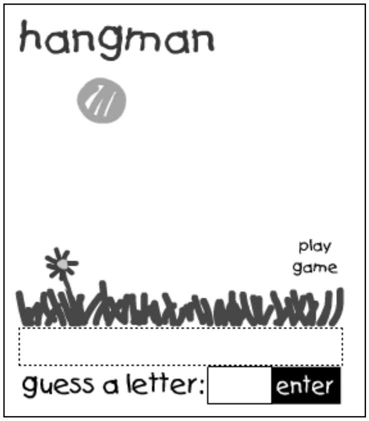
The main movie timeline

This timeline needs two layers, movie and actions, with just one keyframe on each:



- movie will contain the following:
  - An instance of the `title` graphic, which reads "hangman"
  - An instance of the `hanging man` movie clip called `hangman_mc`
  - An instance of the `interface` movie clip called `interface_mc`
  - An instance of the `play` button called `playGame_btn`
- actions will hold all your ActionScript.

Here's how the stage should look once everything's in place:



Well, you've got all the pictures in place—how about a little code? Now the code for this game might look a little hairy at times, but don't panic! Nothing you do here will be hideously complicated. If it all starts to look like a mess of brackets and semicolons swimming about in front of your eyes, just take it one line at a time. Look for the structures you understand and think how they're being used to form the game described earlier on.

Adding the ActionScript

Select frame 1 on the actions layer, open the Actions panel, click that pushpin button, and get ready to wire everything together. Although there are lots of actions to put in over the next couple of pages, they all need to be attached to the same frame. What's more, they break down into three neat pieces:

- **Initialization**: Actions to run when the movie starts
- **Game setup**: Actions to run when you click play game to start the game
- **Game play**: Actions to run when you click enter to submit a guessed letter

Let's get started!

1. The first chunk of code you need to add will set your movie up when it initializes. You begin by creating three arrays:

```
// MAIN INITIALIZING CODE...
// RUNS WHEN MOVIE STARTS
//
// initialize array of words
var wordList_array:Array = new Array("computer", "apple", "coffee",➥
  "flash", "animal", "america", "panel", "window", "import", ➥
  "number", "string", "cake", "object", "movie", "aardvark", "kettle");
// initialize arrays for letters
var lettersNeeded_array:Array = new Array();
var lettersGuessed_array:Array = new Array();
```

The first of these arrays lists all the words you're going to choose from. If you want, you can add more by simply sticking them on the end of the list. Words must be in lowercase letters (since that's all you'll allow the user to enter).

You're going to be using the elements of lettersNeeded_array to store letters from your mystery word, while lettersGuessed_array array elements will hold a mixture of question marks and letters that the player has guessed correctly—these are the letters you'll show the player while he's playing the game. Why do you need two arrays? Well, there are two versions of the same word you want Flash to use: one that it keeps to itself and one that it shows you.

For example, if the word to be guessed was “apple,” lettersNeeded_array would contain the letters a, p, p, l, and e, whereas lettersGuessed_array would start off as ?, ?, ?, ?, ?, and become ?, p, p, ?, ? if the user guessed “p.”

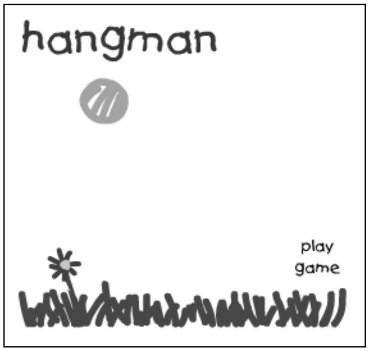
2. Next up, you need to initialize all the variables you use. Rather than explain each here, we’ll explain them as you see them being used in anger once you get to the main code.

```
// initialize variables
var randomNumber:Number = 0;
var selectedWord_str:String = "";
var lettersLeftToGo:Number = 0;
var foundLetter:Boolean = false;
var notGuessed:Boolean = false;
var wrong:Boolean = false;
```

3. You then have to stop the main timeline, stop the hangman_mc timeline, and hide the game interface:

```
// hide the game interface
interface_mc._visible = false;
// stop the hangman and main timelines
hangman_mc.stop();
stop();
```

At this point in the game, all the basic data has been defined, and the game is all set up waiting for some user interaction. The screen will look like this after the main initialization code described in steps 1 and 3 has run:



Now that you’ve set up the movie with arrays and stopped timelines, it’s time you laid down some actions that will get you ready for a game.

Remembering that event handlers go before the main code, the rest of the code goes in front of the lines you’ve just added.

4. When the player clicks the play game button, there are several commands you need to give Flash before you can let the player start guessing letters. You can put these into an event handler for the onRelease event of the playGame_btn instance. It’ll start out like this:

```
playGame_btn.onRelease = function() {
  // initialize graphics
  hangman_mc.gotoAndStop("play");
  playGame_btn._visible = false;
  interface_mc._visible = true;
  interface_mc.display_txt.text = "";
```

First, you initialize the hangman_mc movie clip, sending its playhead to the first frame. In fact, at this point in the game, it's probably there already, but you'll want to run these actions again when you set things up for a rematch, so this is an important thing to do.

You then make the play game button invisible (since you don't need it now) and make the game interface visible (you're certainly going to need that!).

5. Next up in the event handler, you select a word at random from the wordList_array array:

```
// select a word at random & count how many letters it has
randomNumber = Math.round(Math.random()*(wordList_array.length-1));
selectedWord_str = wordList_array[randomNumber];
```

At first glance, you may think the first line in the preceding code looks pretty terrifying. All it actually does is pick a random whole number between 0 and wordList_array.length-1, and store it in a variable called randomNumber. You needn't worry too much about the random function right now, as we'll cover it in a little more detail in Chapter 8.

On the last line, you then use the random number to specify one of the elements in the wordList_array array, pull out the word it contains, and store this randomly selected word in selectedWord_str.

6. The next step is to set up a variable that's going to help you keep track of how many more letters the player still has to guess. At this stage, the player has guessed none, so this is just the number of letters in the word. Assuming the randomly selected word chosen was "apple," lettersLeftToGo would be 5.

```
  lettersLeftToGo = selectedWord_str.length;
```

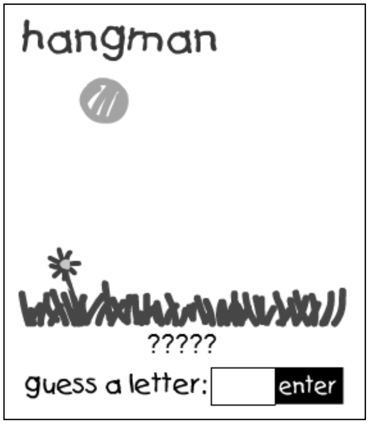
7. Now it's time to break your word down into a list of letters. As mentioned before, the lettersNeeded_array array is going to store all the letters your player has to guess. You set this up using a for loop and cycle through each letter in the word, putting it into the corresponding element in the array:

```
  for (var i:Number = 0; i < selectedWord_str.length; i++) {
    lettersNeeded_array[i] = selectedWord_str.charAt(i);
  }
```

The charAt(i) bit at the end of the second line is new, but all it actually means is "Get the character at position i" of the selectedWord_str variable it's stuck on the end of. This loop would take your word "apple" and turn it into the letters "a," "p," "p," "l," and "e."

8. You do a similar thing for the lettersGuessed_array array but fill every slot with a question mark—this is only fair, since the player hasn't guessed any letters yet!

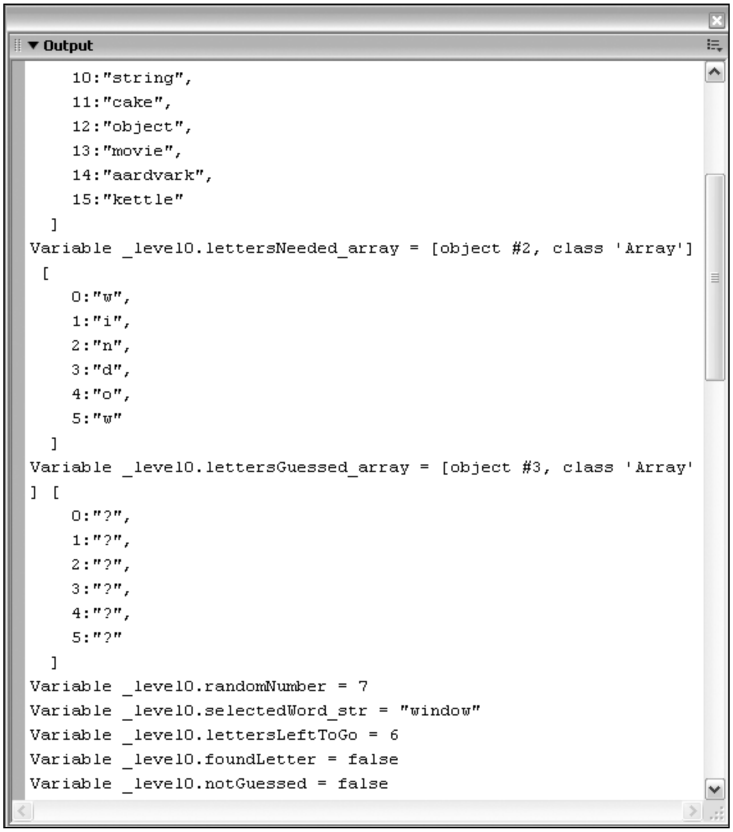
```
  for (var i:Number = 0; i < selectedWord_str.length; i++) {
    lettersGuessed_array[i] = "?";
    interface_mc.display_txt.text += "?";
  }
};
```



Steps 4 to 8 describe the entire script that runs when you click the play game button and get to the following screen. The array `lettersGuessed_array` is the one that's being displayed in `interface_mc.display_txt`, the top text field. Although Flash knows what the letters of the word to be guessed are (as held in the array `lettersNeeded_array`), it's showing a string of the same length but currently consisting of ?s.

Phew! Let's stop for a moment and take a breather. That's a lot of script and a fair number of new tricks along the way. Try running the movie now and click the play game button. Sure enough, the button vanishes, the interface appears, and a string of question marks appears just below the grass, as in the image to the right.

Now select Debug ➤ List Variables, and you can check out the contents of all your arrays:



```
    10:"string",
    11:"cake",
    12:"object",
    13:"movie",
    14:"aardvark",
    15:"kettle"
  ]
Variable _level0.lettersNeeded_array = [object #2, class 'Array']
 [
    0:"w",
    1:"i",
    2:"n",
    3:"d",
    4:"o",
    5:"w"
  ]
Variable _level0.lettersGuessed_array = [object #3, class 'Array'
] [
    0:"?",
    1:"?",
    2:"?",
    3:"?",
    4:"?",
    5:"?"
  ]
Variable _level0.randomNumber = 7
Variable _level0.selectedWord_str = "window"
Variable _level0.lettersLeftToGo = 6
Variable _level0.foundLetter = false
Variable _level0.notGuessed = false
```

> *As well as helping to reassure you that things are going according to plan, viewing your data in the* Output *window should also serve as a reminder that the information you store inside a Flash movie isn't that hard to pull out again. Sure, once you've compiled a SWF and set it loose in the outside world, it won't be quite this easy to peer inside at what's going on, but it's certainly not impossible. Anyone with a bit of tech-savvy and enough perseverance can figure out exactly what words you've got stored in* `wordList_array` *and cheat you out of a victory. You have been warned!*

That aside, you've got a game to finish and an enter button to wire up. Let's get to it!

9. The enter button (instance name `enter_btn`) is actually *inside* the movie clip instance called `interface_mc`, so you need to set up your event handler like this:

   ```
   interface_mc.enter_btn.onRelease = function() {
   ```

   Suffice it to say, `interface_mc.enter_btn` just means "the instance called `enter_btn` that's **inside** the instance called `interface_mc`." You'll learn a lot more about these dot-based shenanigans in the next few chapters, so don't let this oddity put you off just yet!

10. The first thing Flash has to do once the player clicks enter is assume that the player has guessed wrong and also clear the displayed text (because you need to redisplay the guess so far, replacing any ?s that have been guessed with the appropriate letters).

    ```
    wrong = true;
    interface_mc.display_txt.text = "";
    ```

    Now you just loop through all the letters in the word (as stored in the `lettersNeeded_array` array) and change `wrong` to `true` as soon as you find a match with the input text field in `interface_mc`:

    ```
    for (var i:Number = 0; i < selectedWord_str.length; i++) {
      foundLetter = lettersNeeded_array[i] == interface_mc.input_txt.text;
      if (foundLetter) {
        // guess matches a letter
        wrong = false;
        lettersLeftToGo--;
        lettersGuessed_array[i] =  interface_mc.input_txt.text;
      }
      interface_mc.display_txt.text += lettersGuessed_array[i];
    }
    ```

    The variable `foundLetter` gets its value from `lettersNeeded_array[i] == interface_mc.display_txt.text`, which will always be a Boolean expression. It will be true only if the letter your player just entered is the same as the letter in element `i` of the `lettersNeeded_array` array.

    You also bump down the number of letters left to guess, change the appropriate ? in the `lettersGuessed_array` array so that it shows the correct letter instead of a ?. At the end of each loop, you update the displayed text (`display_txt`) so that it reflects the fact that some of the ?s may now have been guessed correctly.

Hang on a minute, though! What if you picked the same letter more than once (such as guessing “p” in “apple”)? If it matched the first time, it would match again and again; lettersLeftToGo would hit 0 before very long, suggesting that you’d won the game! It looks like you need to be a little more picky with the condition in your if statement.

11. Make the following highlighted additions to the for loop you just wrote:

```
for (var i:Number = 0; i < selectedWord_str.length; i++) {
  foundLetter = lettersNeeded_array[i] == interface_mc.input_txt.text;
  notGuessed = lettersGuessed_array[i] != interface_mc.input_txt.text;
  if (foundLetter && notGuessed) {
    // guess matches a letter we haven't already found
    wrong = false;
    lettersLeftToGo--;
    lettersGuessed_array[i] = interface_mc.input_txt.text;
  }
  interface_mc.display_txt.text += lettersGuessed_array[i];
}
```

Now you check that your player’s input is *not* the same as element number i in the array lettersGuessed_array. Now, if foundLetter and notGuessed are *both* true, you know you’ve got a brand-new correct guess!

12. Next, you empty out the input box, so that it’s all ready for your player to try again:

```
// reset input text box
interface_mc.input_txt.text = "";
```

13. If the player’s guess was no good (or the player was foolish enough to guess at a letter he or she had already gotten), the variable wrong will still have a Boolean value of true, in which case the next few lines kick in:

```
if (wrong) {
    hangman_mc.nextFrame();
    if (hangman_mc._currentFrame == 10) {
        // GAME OVER!!
        interface_mc._visible = false;
        playGame_btn._visible = true;
    }
}
```

First, you use nextFrame() to bump the hangman_mc playhead onto the next frame. You then check whether that happens to be frame 10, also known as HANGMAN!

If it is frame 10, you run the “Game Lost” actions. These just hide the interface (the text fields and the enter button) and show the play game button (so the player can play the game again if he or she feels the need). If it’s not frame 10, you don’t need to anything here.

14. Last bit now! You just need to test whether all the letters have been guessed correctly. If they have, the value of lettersLeftToGo should be down to 0. You use this fact to control whether or not the "Game Won" actions should be run:

```
    if (lettersLeftToGo == 0) {
        // GAME WON!!
        hangman_mc.gotoAndStop("win");
        interface_mc._visible = false;
        playGame_btn._visible = true;
    }
};
```

These simply send hangman_mc to the win frame (where you get to see your hero holding a flower and smiling in gratitude), hide the interface, and show the play game button.

Now that you're all done, you can sit back and relax—and maybe have a quick game! Of course, the tension won't be quite so high now that you've seen the full list of words your movie gets to choose from. All the same, though, it's nice to see that all this script will actually do something.

> *Also worth looking at is* hangman_advanced.fla*. This FLA prevents you from entering the same letter twice (using the rather handy* restrict *property of a text field) or a blank space as your guess.*

Book project: Creating the static graphics

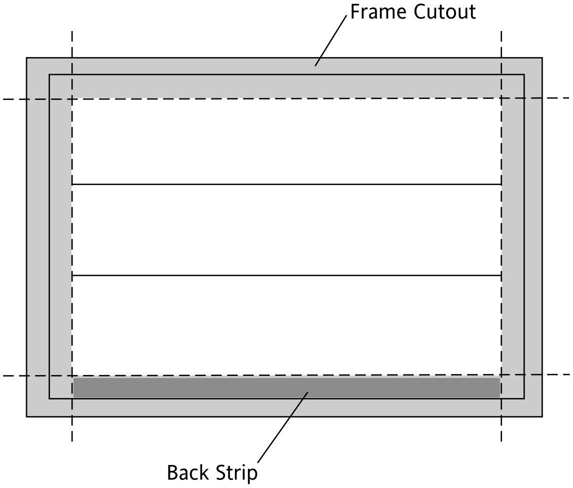
You're not yet at the point of writing any ActionScript in the book project, and you're still in the process of creating the graphics.

In this section, you'll create the graphics in the border area. Not only do the graphics you're about to create have no ActionScript associated with them right now, but also they *never* will because they're **static**—they just sit there looking pretty.

You may now be thinking, "Wha . . .? I thought this was a book on ActionScript!" It is. It's just that coding up a Flash site—even one that's heavily ActionScript oriented like Futuremedia—isn't just a straight programming exercise. There's a strong graphical aspect to it. A bit of a code respite will probably also soothe some jangled nerves after that hangman game!

> *You may be coming into this book with JavaScript or other web programming experience, or perhaps you've spent some time already with Flash and just not "gotten it." If this is you, don't worry.*
>
> *The thing that makes ActionScript so different from other programming languages is the closeness of the relationship between the code and graphics. Unlike what you may be used to, a significant portion of your setting up in Flash has to do with graphics and layout rather than code. This is to be expected. Despite recent advances in the ActionScript language, Flash is still predominantly a visual environment. Unlike JavaScript, whose most common use is browser control or client-side validation, the main use of ActionScript is animation. For that to take place, you first have to have some graphics.*

As always, you'll first review what you'll be doing this chapter section through the use of a trusty rough sketch. You'll create the back strip at the bottom of the site plus another hidden graphic that you didn't notice when you looked at the finished site: a frame that sits around the border of the site.



The function of the back strip is obvious when you look at the final site: it's a design widget that pulls together all the parts of the interface that sit at the bottom of the screen.

What does the frame cutout do? Well, rather like a puppet-show booth shows the puppets but hides the puppet master, you want the user to see what's going on during the zoom transition, but you want to hide all the tricky stuff you're doing behind the scenes, and this is what the cutout does. It hides what's *really happening* by hiding the portions of the screen that you don't want the user to see. As the three-color stripes scale up, they would start to fill the border area, but you stop this from happening by placing the frame cutout in front of them.

In short, the frame cutout prevents the viewable site from changing size as the zoom trick takes place. And yes, you can't see the frame cutout in the final site, and that's because it's the same color as the stage and HTML background. Despite being hidden, it's crucial to the final effect and helps give the impression that the navigation is more complicated than it actually is (a designer trying to deconstruct the effect using code only would fail without the frame cutout).

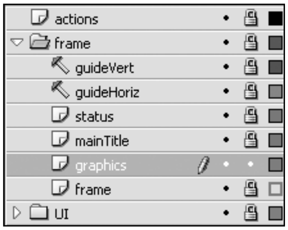
Some of the trick that makes most Flash ActionScript sites cool is code, but some of it comes down to graphic design. As noted at the beginning of this section, not all the clever stuff that happens in Flash is related to the code!

The starting point of this section is the work in progress from Chapter 3. If you don't want to use your own file (or you get stuck and want something to compare notes with), you can use the file `futuremedia_setup2.fla`.

OK, that's the all-important preamble out of the way. Let's get down to some Flash authoring!

Setting up the timeline layers for your graphics

In the frame layer folder, add two new layers called graphics and frame. Lock all layers except graphics.

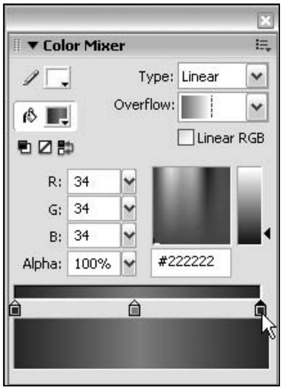


Creating the back strip

Follow these steps to create the back strip.

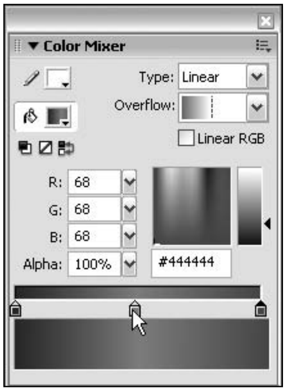
1. Although the back strip looks black, it actually has a subtle gradient on it to stop it from looking flat. The first step is to create this gradient. Using the Color Mixer (Window ➤ Color Mixer), create a linear gradient for the fill color as shown. The gradient has three control colors.

   The two edge colors are almost black: R: 34, G: 34, B: 34, Alpha: 100% (or #222222 for the HTML heads).



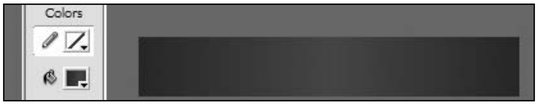
Type: Linear
□ Linear RGB
R: 34
G: 34
B: 34
Alpha: 100%
#222222

   The center color is a slightly lighter shade of almost-black: R: 68, G: 68, B: 68, Alpha: 100% (or #444444).

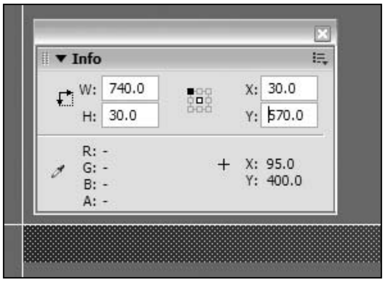


Type: Linear
□ Linear RGB
R: 68
G: 68
B: 68
Alpha: 100%
#444444

2. Select layer graphics. On the stage, draw out a long black oblong with no stroke. Don't worry about the size or position too much or the fact that the gradient doesn't look right. You will fix all that almost immediately.



3. Bring up the Info panel (Window ➤ Info). Select the oblong and set the registration point on the Info panel (via the little matrix in the following screenshot) to the top left. Set the width to 740 and the height to 30. Set the position of the oblong to X: 30, Y: 570. This will move and scale the oblong so that it fills the lower edge of the border as shown:

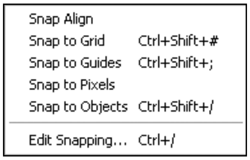

Info

W: 740.0 X: 30.0
H: 30.0 Y: 570.0

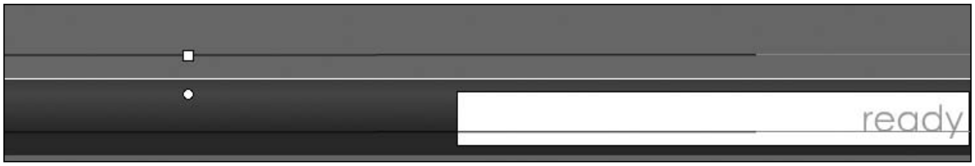
R: -
G: -
B: -
A: -

X: 95.0
Y: 400.0

4. The gradient runs vertically, but you want it to run horizontally. To fix this, you need to use the Gradient Transform tool. You'll do some close scaling in a moment, and although Flash will want to lend a helping hand, you don't need it, so make sure all snapping options in the View ➤ Snapping submenu are unchecked.


Snap Align
Snap to Grid Ctrl+Shift+#
Snap to Guides Ctrl+Shift+;
Snap to Pixels
Snap to Objects Ctrl+Shift+/
Edit Snapping... Ctrl+/

5. Deselect the back strip by clicking any other area of the stage, and then reselect it with the Gradient Transform tool. Rotate the fill by 90 degrees (it doesn't matter in which direction; either clockwise or counterclockwise is fine). Scale the envelope until it's about the same height as the back strip. Finally, move it up slightly so that you end up with an envelope whose lines run parallel with the long edges of the strip and whose center point is about one-fifth of the way down from the top edge of the back strip, as shown:

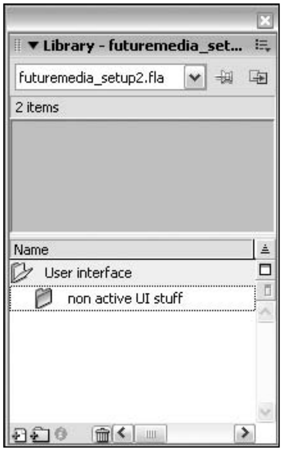


Adding structure to the Library

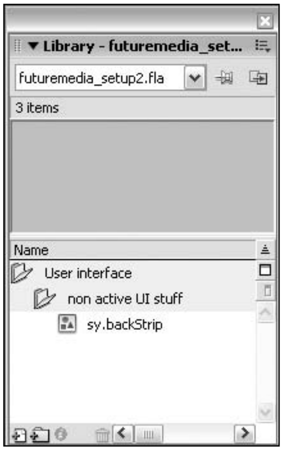
OK, so now you have a primitive (i.e., it isn't yet a Flash symbol) rectangle on the screen. You should really make it into a symbol, but before doing that, let's create some structure for the (many) symbols that you'll be adding later on. This includes two things:

- A well-defined library folder structure
- A well-defined naming structure for your symbols

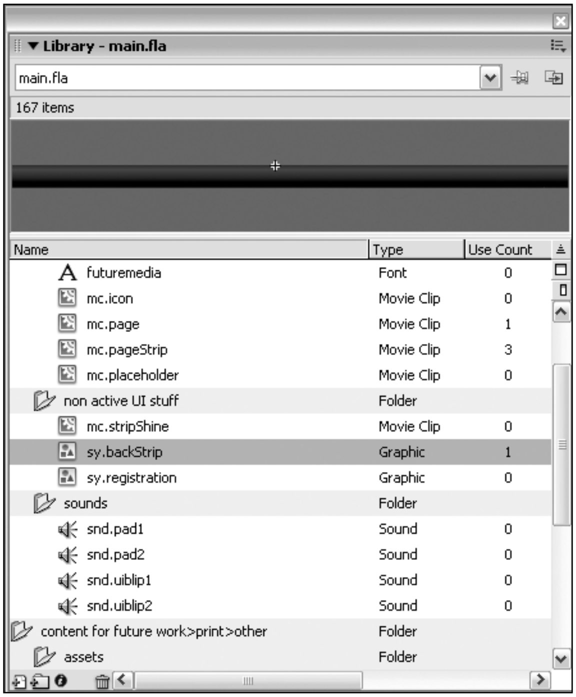
1. In the Library pane (Window ➤ Library or Ctrl+L/Cmd+L, but you knew that already, didn't you?), create a new folder called User interface. Inside that folder, create another folder called non active UI stuff.



2. Now with the back strip selected, press F8 and make the back strip a graphic symbol with the name sy.backStrip.



So why do you call your symbol something with sy at the front? Well, the Flash Library sorts by alphabetical order by default. Adding your own prefixes forces Flash to sort by some criteria you set. You can see this in a typical shot of the final site.



167 items

Name	Type	Use Count
futuremedia	Font	0
mc.icon	Movie Clip	0
mc.page	Movie Clip	1
mc.pageStrip	Movie Clip	3
mc.placeholder	Movie Clip	0
non active UI stuff	Folder	
mc.stripShine	Movie Clip	0
sy.backStrip	Graphic	1
sy.registration	Graphic	0
sounds	Folder	
snd.pad1	Sound	0
snd.pad2	Sound	0
snd.uiblip1	Sound	0
snd.uiblip2	Sound	0
content for future work>print>other	Folder	
assets	Folder	

As you can see, things are going to get complicated around your currently solitary sy.backStrip! The way we use prefixes forces Flash to sort the symbols so related groups are listed next to each other.

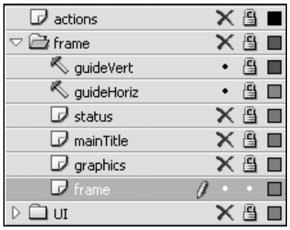
Flash allows you to do something very similar without using the prefixes. If you click the Type *column, the Library will sort on symbol type. That's great, but sometimes you want to subdivide further, such as between* mc *(standard movie clip),* ma *(movie clip containing only ActionScript), and* mp *(movie clip placeholder).*

Getting in the habit of using meaningful prefixes helps immensely when you're building very large Flash sites, but you'll find that it works even in smaller sites. It's amazing how many symbols even a modest site creates. The finished version of Futuremedia contains a few hundred symbols, but you couldn't count that many just by looking at the finished site!

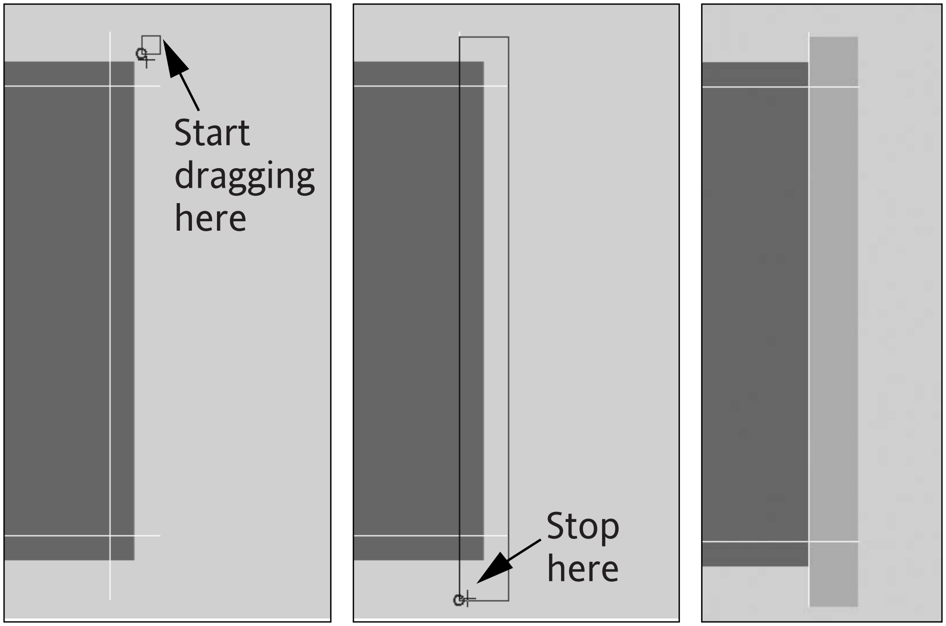
That's the first of your graphics done—one more to go.

Creating the frame cutout

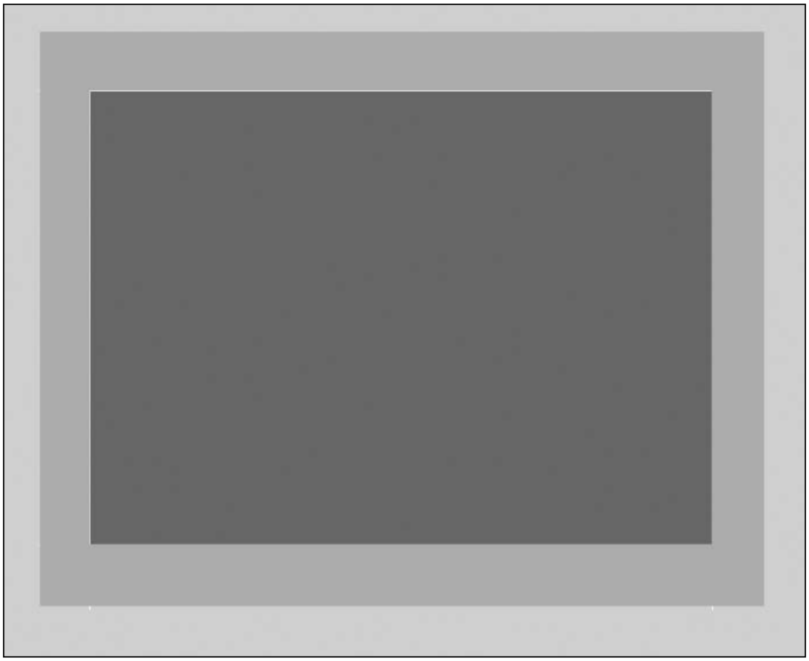
Follow these steps to create the frame cutout.



1. Lock layer graphics and unlock frame. Hide all layers except the two guide layers and layer frame. Select layer frame.
2. Select the Rectangle tool with no stroke and a fill color that contrasts with the background (such as bright green).
3. Select View ➤ Snapping ➤ Snap to Objects, and draw out a rectangle starting from the top-right corner and going off the stage, down toward the bottom right, making sure that it covers the border gutter and some of the area off the stage (see the following diagrams). As you get close to the guide lines you added, the cursor will snap to them. You want the rectangle to go up to (but not beyond) the guide lines, so that the rectangle covers only the border.



4. Repeat until you have a hollow frame shape (rather like a picture frame) that covers all the border areas and extends a little beyond the stage edges on all four sides.

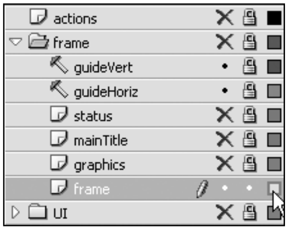


5. Use the Fill tool with the fill color set to the background color (#666666, or use the Eyedropper tool to sample the background color), and then change the frame cutout color to this color.
6. Group the frame with itself (select it and press Ctrl+G/Cmd+G or select Modify ➤ Group). That may seem a little strange—why group a graphic when there's only one? Well, if you were to later inadvertently draw over the frame with other graphics, they would tend to split your frame or cut shapes out of it, unless you group the graphic or make it into a symbol. There's no need to make the graphic into a symbol because there will only ever be one (and making symbols when you don't need to only clutters up the Library). So instead, you just group it with itself.

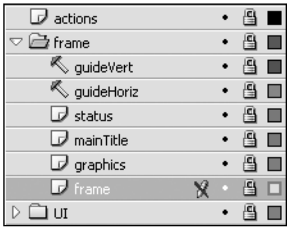
Revealing the frame cutout and finishing the FLA

The frame is now well hidden, given that it's the same color as the stage. It now looks like part of the stage (which is exactly what it does in the final site). That's great, but it will only confuse you in the authoring environment as you develop your site further. The way around this is to make your shape appear in a different way while you're editing the FLA.

1. Make the frame layer show outlines only by clicking the colored brick for this layer.

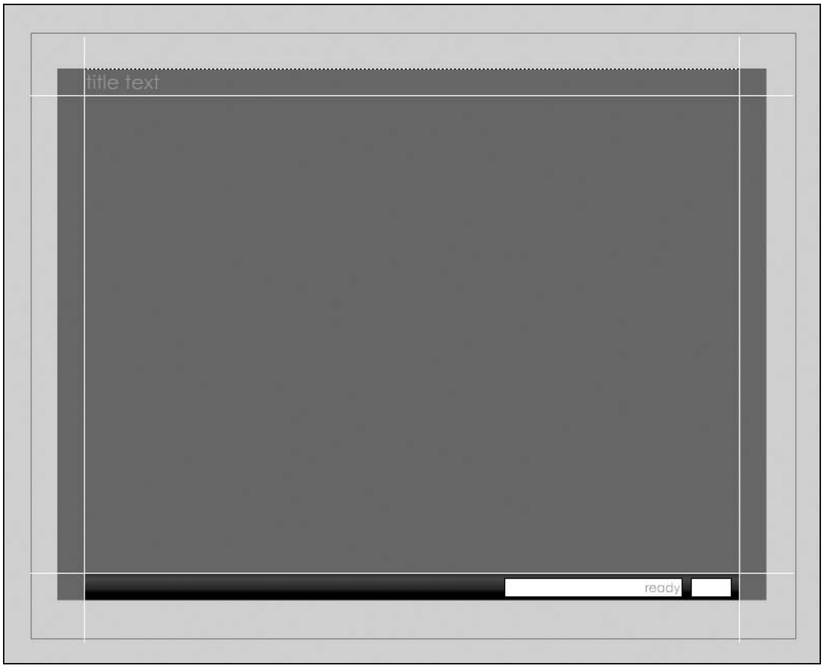


2. To finish up, show and lock all layers, and then save your file.



Parting shots

So, you now have a FLA that looks only marginally less boring than the one at the end of Chapter 3.



Well, that's certainly one way of looking at it, but you've started off slowly because you're following a *design process*. You already know how the final site will work in principle. You know which parts you've done so far (and you can probably guess at what's outstanding).

More important, you've learned that code alone doesn't make an ActionScript site. You have to set up the graphics first or at least give them some thought before you plow into code:

- That little dumb back strip may have only taken a few minutes to create, but it's something that has a large effect on your final site's look and feel.
- The frame cutout didn't take long to create either, but the thought process I had to go through to create it was significantly longer. I had to consider alternative implementations such as using masking (too processor intensive) or doing away with the border altogether (which would prevent adding any icons or other controls around the edges later on, making the design less flexible).

These thoughts, tests, and design decisions were worked out without using any code. The "animate or draw it manually in Flash and see what the issues are as you move the whole interface by hand" trick was used for the code-centric problems, and the back strip was designed outside Flash (it was mocked up in Photoshop, where the initial colors for the final UI were also designed).

Although these tests and issues aren't included in the book, you're still at the nonprogramming stage and currently working it all out using standard Flash drawing tools, sketches, and basic graphic design.

> *To put all this into some sort of perspective for the beginner who is possibly still thinking "Where did he get all these ideas from? I would never have come up with any of this!" it took almost three days to get to the point where I knew what I was doing would work and had a good idea how to do it in principle, and I had completed the final FLA to the stage you've just reached. You'll see the amount of time you spend go down rather than up as you start adding major portions of code to the site, because you thought through the design at this initial stage.*
>
> *—Sham Bhangal*

Now that you've completed the static parts of the UI, you'll look at the moving parts—the **dynamic graphics**—in the next chapter. After that, you'll get down to some *serious* scripting. By the end of the book, you will have a 400-odd-line script to control this currently rather bleak-looking FLA. So don't get complacent by this slow initial pace. It's going to get busy around here real soon.

Summary

In the course of this chapter, you've looked at three different ways to make something loop in Flash:

- Timeline loops using the `gotoAndPlay()` action
- ActionScript loops using the `while()` action
- ActionScript loops using the `for()` action

That's not to say there are no other looping actions—there's `do...while`, which is almost exactly the same as `while`, except that it tests its condition *after* running the loop actions each time. There's also `for...in`, which you can use to loop through all the elements in an array, the properties of an object, or all the instances in a timeline. When all's said and done, though, they don't offer much beyond what you've already achieved. As you've seen, even `for` loops are really just a specialized version of your basic `while` loop—just a particularly useful one! We haven't set out to try and be comprehensive here, since the three loops covered in this chapter should do for most of your needs. The only question is when you should use each one.

Here are some simple rules of thumb for deciding when (and when not) to use an ActionScript loop or a timeline loop:

- If you want Flash to perform an action (or set of actions) many times over in the shortest possible period of time, you should use an ActionScript loop. Every cycle of the loop is attached to one frame, so (unless you run over 200,000 cycles, when Flash starts getting a little nervous and decides your code has an infinite loop in it) they'll always complete before you move on to the next frame.
- If you want to make something move gradually across the stage, you're best off using a timeline loop (such as an `onEnterFrame` event) to nudge it by a few pixels each frame. An ActionScript `for` or `while` loop will work its magic much, much faster, so you're unlikely to see the start and end points of the motion.

If you've settled on an ActionScript loop, picking between the `while` and `for` loops is fairly simple:

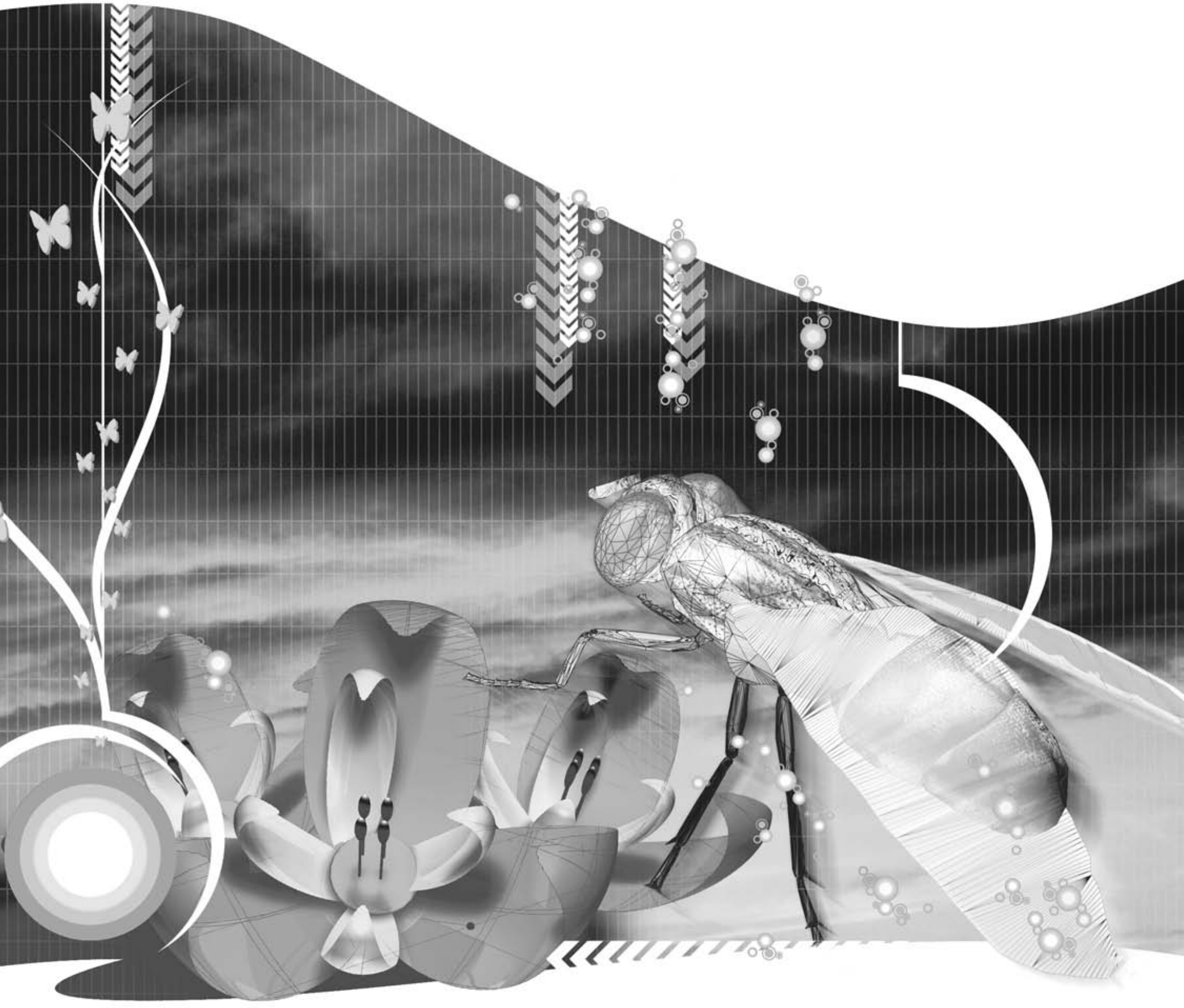
- If the values you use to define the looping behavior are all known in advance, then it's best to use a `for` loop. Otherwise, a `while` loop will almost always do the trick.

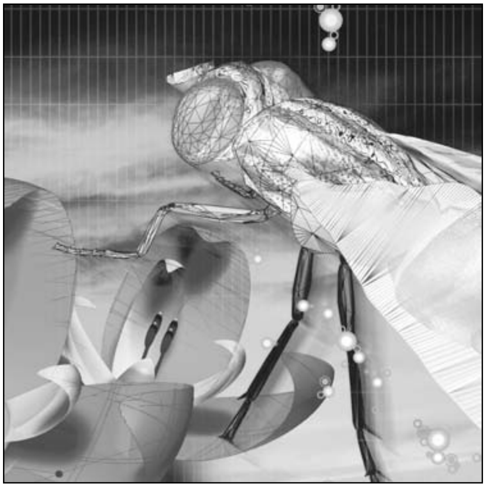
This chapter has had the biggest variety of examples so far, because the full range of the ActionScript skills you've been learning is really starting to come together. You've learned some important new techniques, and even more important, you've seen large sections of ActionScript working together in a finished application.

Don't worry if you've only skimmed through this chapter. The important thing is that a standard has been set in your mind, and you've probably picked up a lot of useful little snippets without even realizing it. That's what coding is all about. You're probably looking at all the long pieces of code and thinking, "Why has he done that? Why didn't he do it this way?" The proof is when you start doing your own programs and recognize things you're trying to do in terms of the programs listed here.

Each of the last three chapters has been an important step forward, building up the range of things you can do once you start throwing information around behind the scenes of your movies. Now that you've seen how to work with variables, arrays, decision making, and looping, you should be getting a very good idea of just how much a few lines of ActionScript can add to your Flash movies.

You're not even halfway through the book yet, and you're already moving away from the level of novice ActionScripter. In the next few chapters, you're going to look at more ways of bundling together actions and information, and you'll start thinking about how your scripts are structured. You may already feel confident enough to wire the odd snippet of ActionScript into your own projects—if so, that's great! Soon enough, though, you'll be all set to start taking on some full-sized challenges.





Chapter 6

MOVIES THAT REMEMBER HOW TO DO THINGS

What we'll cover in this chapter:

- Breaking down tasks into subtasks and using functions to structure your code
- Understanding the difference between anonymous and named functions
- Using functions to recycle blocks of script
- Using arguments and returning results
- Making functions more robust through typing
- Building the dynamic graphics for the Futuremedia site

As you've seen, the ability to handle information in your movies opens up a lot of possibilities. A few chapters back, we introduced variables as containers for data, whether that data is in the form of a string, a number, or a true or false Boolean value. We also covered arrays, which let you store lots of variables under a single name.

Variables and arrays help you to store and organize your data, but data is only half of the deal when it comes to writing scripts. What about actions? Literally anything you do with ActionScript is going to involve at least one action (or command) giving Flash an explicit instruction to do something or other. That's the whole point of ActionScript—of course, that's why it's called *Action*Script in the first place!

This is where functions come in. Say you have a sequence of actions you need to use at several different points in a movie. Rather than writing them out in full each time, you can write a **function declaration** that contains these actions, and then call out for that function whenever you want the actions to run. You just need to give Flash the name of the function, and it will track down the function definition for you and make sure it runs everything inside.

In many ways, a function does for actions what a variable does for data: it acts as a container for things you want to reuse later in the movie.

This turns out to be very useful for organizing your scripts too. Functions let you wrap whole blocks of script into easy-to-handle sections, and if you pick those script blocks wisely, you'll ensure that each one focuses on a specific subtask.

Imagine that you want identical functionality in different parts of your application. By organizing the necessary code in a function, if you ever need to make any changes to the way the code works, all you need to do is change the function, rather than track down code all over the place. This allows for more modular code, easier maintenance, and easier readability.

Breaking down a task

Think about the whole business of writing a script in everyday terms. Whatever you happen to be working on, you're essentially trying to break down one complex task into several simpler ones.

When you go shopping, there are lots of actions you'll perform as part of the overall task: walk to the store, find the items you want to buy, take them to the checkout counter, find out the total cost, hand over a sum of money, receive change (hopefully!), leave the store, and walk home with your items.

This is all so obvious that you barely need to think about it. If someone asks you to fetch something from the store, you already know what that's likely to entail. Of course, the person will need to tell you what to buy, and possibly which store to buy it from, but the basics of "go shopping" are already there.

Most of the examples you've looked at so far have involved taking an overall task and breaking it straight down to the level of individual actions. This works well for the small applications you've been dealing with, but once your programs start to grow in size, things can quickly get out of hand. As we're about to demonstrate, functions give you an opportunity to break down your overall movie mission into tasks and subtasks just as quickly or slowly as you want to.

There are two very good reasons for using functions to wrap up your code into blocks:

- **Reusability**: You can use each block over and over again with the minimum of fuss.
- **Structure**: You can break up huge reams of code into easily identifiable sections.

Now that you know why functions are a good thing, let's look at how you can actually use them.

Bundling actions and running them afterward

Working with functions isn't hard at all, especially in light of what you've already done since the beginning of this book. There are really only two steps involved:

- **Declare** the function by giving it a name and telling it what actions to perform.
- **Call** the function by telling Flash to summon it and run those actions.

Using anonymous and named functions

All of the functions that you have used so far in this book have been attached to event handlers, such as `onRelease` or `onEnterFrame`, like this:

```
enter_btn.onRelease = function() {
  outputNum = inputNum * 2;
};
```

These are called **anonymous functions**, because they have no name. Sometimes you may also see them referred to as **function literals**. Because you don't need to give this type of function a name, they're a quick-and-easy way to assign a set of actions to an event handler. At the same time, unless a function has a name, it's impossible to reuse it in other parts of your script. Although you can assign this sort of a function to a variable, it's much simpler to create a named function instead.

Say you were writing a monster-attack game and wanted to make all the monsters invisible in one fell swoop. You could create a named function called `hideAllMonsters()` and use it over and over again every time you want the monsters to disappear. It would probably contain a loop that cycles through all relevant values of counter variable `i`, making monster number `i` invisible each time around. Here's what the function definition might look like:

```
function hideAllMonsters() {
  for (var i:Number=0; i < numberOfMonsters; i++) {
    _root["monster"+i+"_mc"]._visible = false;
  }
}
```

The basic pattern for creating a named function is very simple:

```
function functionName() {
  // actions go here;
}
```

You begin with the `function` keyword, followed by the name of the function and a set of parentheses. The actions that you want the function to perform go between a set of curly braces.

A strange quirk of ActionScript is that you put a semicolon after the closing brace of an anonymous function, but not of a named one. This is because an anonymous function is assigned to an event handler or variable, so it's part of a statement, which should always end with a semicolon. A named function, on the other hand, stands on its own.

When it comes to picking a good name for a function, the issues are similar to those you looked at for naming variables. You should make the name relevant and recognizable, and take care not to use one of Flash's reserved words.

So what about *using* these function-thingies, then? Simple. Use them just as if they were actions. Here's how you might call your monster-hiding function from elsewhere in the script:

```
if (monsterCamouflageOn) {
    hideAllMonsters();
}
```

You can even think of a function as being like your own custom action—one that's totally exclusive to your movie!

Using functions to hide evil math

Let's start with a very simple example. Say you're writing some ActionScript within a Flash movie that has to perform an evil bit of math on a variable called `myVariable`. It's not nice, but it's sometimes necessary to get the jaw-dropping effects you're after. OK, you'll hold back on the *really* evil math for now and just do something like "add 5 and multiply by 2." That seems simple. You're probably already thinking of something along the lines of this:

```
// input number
var myVariable:Number = 4;
// add five
myVariable += 5;
// double it
myVariable *= 2;
// show the result on the screen
trace(myVariable);
```

This code introduces two new operators: `+=` and `*=`. They're useful bits of shorthand known as **combined operators** that are used by most programming languages. Combined operators consist of an arithmetic operator, which performs a calculation on the variable; and the equal sign, which assigns the result back to the same variable. If you've never come across this before, it takes a little getting used to, Take a look at the following line of code:

```
myVariable += 5;
```

It means exactly the same as this:

```
myVariable = myVariable + 5;
```

Similarly, take a look at this line of code:

```
myVariable *= 2;
```

It means exactly the same as this:

```
myVariable = myVariable * 2;
```

Moving back to the original calculation of "add 5 and multiply by 2," what if you need to perform exactly the same calculation another 20 times in various places throughout your movie? Of course, you could just type out the same code over and over again, but this will quickly get to be a real drag, especially if your calculation runs to more than a couple of simple expressions and you wind up having to type out seven or eight lines every time.

This is the perfect time to use a function. This is how you might do it:

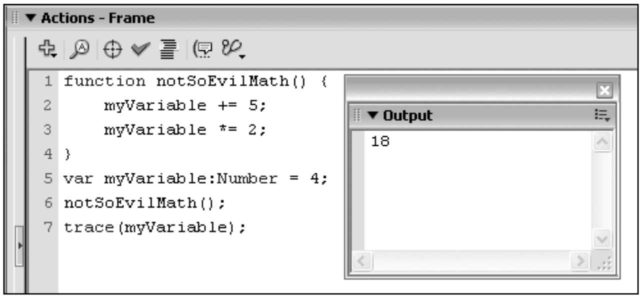
```
function notSoEvilMath() {
  myVariable += 5;
  myVariable *= 2;
}
```

There you are—a simple function to take myVariable, add 5, and double it. Now you can use a notSoEvilMath() function call in place of your original math. Create a new Flash movie and attach this script to the first frame:

```
function notSoEvilMath() {
  myVariable += 5;
  myVariable *= 2;
}
var myVariable:Number = 4;
notSoEvilMath();
trace(myVariable);
```

You should declare all your functions near the beginning of the first frame, before you write any script that needs to call them. Although Flash will normally run a function correctly if it's called before it's been defined, you shouldn't rely on this. It's like taking food out of the oven before it's properly cooked. It might be OK, but you might end up with an unpleasant surprise.

Run the movie, and you should see the Output window pop up with the answer like this:



```
function notSoEvilMath() {
    myVariable += 5;
    myVariable *= 2;
}
var myVariable:Number = 4;
notSoEvilMath();
trace(myVariable);
```

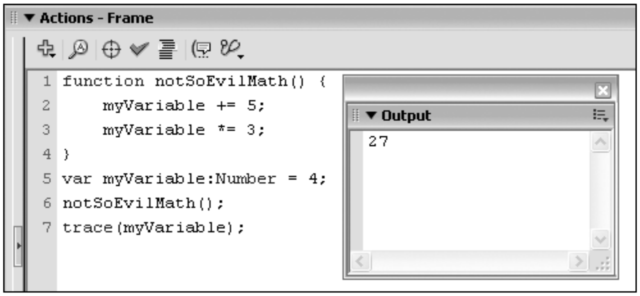
```
18
```

There you are: a simple function declaration and function call.

There's another benefit to using functions here. Say you suddenly find you need to alter the code so that it triples the input instead of doubling it. If you had done it ten times the longhand way, you would need to change ten different lines of script. With the function, you need to change only one character on one line:

```
function notSoEvilMath() {
    myVariable += 5;
    myVariable *= 3;
}
```

Every time it's called now, `notSoEvilMath()` will add 5 and *triple* the value of `myVariable`, just as if that was what it had always done.



```
function notSoEvilMath() {
    myVariable += 5;
    myVariable *= 3;
}
var myVariable:Number = 4;
notSoEvilMath();
trace(myVariable);
```

```
27
```

Once you've used functions a few times, you can't imagine how you ever managed without them! Of course, their usefulness becomes more obvious when you're using them for something practical and when you're using them several times in one movie. Let's look at another situation where functions can help you out.

Using functions to take care of repetitive jobs

Think back to the hangman game you put together at the end of Chapter 5 and the faculty records movie you made in Chapter 4. In both cases, you used `_visible` to make buttons appear and disappear when you wanted. This is a very powerful way to control what's shown on the stage in Flash, and you'll often find it being used to substitute one button (or movie clip) for another.

For example, the hangman movie had a play game button and an `interface` movie clip. You only ever wanted one of them on the screen at any particular time: you needed the interface while you were playing the game, and you needed the button when you weren't playing the game. That meant writing this at one point in the script:

```
playGame_btn._visible = false;
interface_mc._visible = true;
```

and this a little later on:

```
playGame_btn._visible = true;
interface_mc._visible = false;
```

Eagle-eyed readers will have already spotted that these two bits of script are almost identical. The difference boils down to which instance gets shown and which one gets hidden. In fact, even this isn't much of a difference: in both cases, the visible one gets hidden, while the invisible one gets unhidden. You can write this out in script as follows:

```
playGame_btn._visible = !playGame_btn._visible;
interface_mc._visible = !interface_mc._visible;
```

The `!` sign is called a **NOT operator**, and it shows you the opposite Boolean value of whatever comes straight after it. If `playGame_btn._visible` is `false`, `!playGame_btn._visible` must be `true`, and vice versa. In effect, both lines of code now say

```
if it's hidden, show it - if it's showing, hide it
```

with one line of code doing that for each instance. You now have a script that you can reuse whenever you need to swap these two instances, so let's make it into a function:

```
function swap() {
  playGame_btn._visible = !playGame_btn._visible;
  interface_mc._visible = !interface_mc._visible;
}
```

If you add this declaration to your hangman script, you can replace each of the true/false pairs with a quick-and-simple call to the new `swap()` function:

```
playGame_btn.onRelease = function() {
  // initialize graphics
  hangman_mc.gotoAndStop("play");
  swap();
  ...
};
```

```
interface_mc.enter_btn.onRelease = function() {
  wrong = true;
  ...
  if (wrong && notGuessed) {
    hangman_mc.nextFrame();
    if (hangman_mc._currentFrame == 10) {
      // GAME OVER!!
      swap();
    }
  }
};
```

This uses swap() only twice, but it's already making the script shorter and easier to follow. Think how much more of a difference it would make if you had 10 or 20 swaps to do! You can see the updated script in hangman_swap.fla in the download files for this chapter.

Choosing which actions to bundle and where

Say you want a script that tells your movie to go shopping ten times in a row. In the past, you'd write a loop (probably a for loop, since you know it needs to cycle ten times) containing all the actions involved in "going shopping." If there are only two or three of these actions, that's probably fine as is, but what if there are two or three *hundred*? Don't laugh—it's quite possible!

Sure, you can still pop them all inside the for loop, but it will make your script a real pain to follow. Another option is to define a function called goShopping() that contains all the actions to go shopping once. The loop that calls goShopping() ten times over will then look like this:

```
for (var i:Number = 0; i < 10; i++) {
  goShopping();
}
```

That simplifies things here, but what about in the function declaration? Oh, no . . . now *that's* going to go on for six or seven pages, and you have the same problem of readability you had before!

Actually, you've already worked out a way around this particular problem. A "go shopping" task is too big and complex, so you break it down into subtasks. What about this:

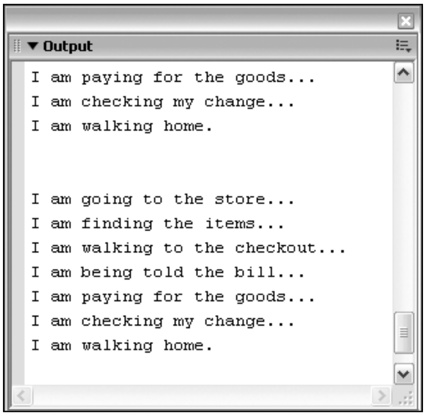
```
for (var i:Number = 0; i < 10; i++) {
  walkToStore();
  findItems();
  walkToCheckout();
  findCost();
  payForGoods();
  checkChange();
  walkHome();
}
```

Of course, you'd have to work out how all these other functions work and declare them as well, but each one will be *much* smaller than the goShopping() declaration would have been.

What you can do at this point, even though you haven't worked out what will go into each function, is use **empty functions** (also called **stubs**) to build up some skeleton code. Create a new Flash document and type the following code into the Actions panel (or use `shopping1.fla`):

```
function walkToStore() {
  trace("I am going to the store...");
}
function findItems() {
  trace("I am finding the items...");
}
function walkToCheckout() {
  trace("I am walking to the checkout...");
}
function findCost() {
  trace("I am being told the bill...");
}
function payForGoods() {
  trace("I am paying for the goods...");
}
function checkChange() {
  trace("I am checking my change...");
}
function walkHome() {
  trace("I am walking home.\n\n");
}
for (var i:Number = 0; i < 10; i++) {
  walkToStore();
  findItems();
  walkToCheckout();
  findCost();
  payForGoods();
  checkChange();
  walkHome();
}
```

The `trace()` actions give you a rough indication of which functions will run when, and in what order. If you test the movie, the preceding listing gives the output shown alongside.



```
I am paying for the goods...
I am checking my change...
I am walking home.


I am going to the store...
I am finding the items...
I am walking to the checkout...
I am being told the bill...
I am paying for the goods...
I am checking my change...
I am walking home.
```

Already you can see the shopping trip coming together.

A listing consisting of empty function stubs is sometimes called a **skeleton**. You write a skeleton consisting of empty function stubs, and then test it with `trace()` messages to make sure it does roughly what you expect. When you're happy with the skeleton, you start fleshing it out by adding code to the functions to get it to do what you really want: move graphics around, act on variables to create an effect, or solve a problem.

The beauty of using functions in this way is that you can make high-level changes early on when your code is still easy to change—at the skeleton stage. This is preferable to making changes at the end when more of your code is written, simply because then there's more that could go wrong.

Also, using a skeleton allows you to avoid using another form of planning that most computer classes seem to teach (but that is actually all but useless in event-driven code): flowcharts.

Arguments and redundancy

Another thing you can spot here is **redundancy**—that is, three of these functions are doing almost exactly the same thing. The only difference between walkToStore(), walkToCheckout(), and walkHome() is the destination they send your imaginary customer walking off to. So is there anything you can do about this? Of course there is!

You've already seen that some actions—gotoAndPlay(), for example—will let you feed in arguments between the parentheses that influence how they're carried out by Flash. Well, you can do exactly the same thing with functions.

Say you set up a general function called walkTo() that lets you feed in a string argument that tells it where you want to walk to. Still sticking with basic stubs, you might declare it like this:

```
function walkTo(destination) {
  trace("I am now walking to " + destination);
}
```

The main difference between this and the other functions you've declared is the fact that you've stuck a variable called destination between the parentheses following the function name. This variable is called a **function parameter**, and you use it to store the value of the argument with which the function was called.

Confused yet? Let's take a step or two back and think about what happens when you call the walkTo() function. Say you write a line of script like this, which calls the walkTo() function using the argument "the store...":

```
walkTo("the store...");
```

Flash sees this, and since it doesn't recognize walkTo() as an ActionScript action, it hunts around for a function called walkTo(). It finds the declaration you wrote and spots that there's one parameter in there, which is called destination. Currently, destination has no value, but that doesn't last long—Flash now takes the argument value "the store..." and stores it in destination, ready for the actions within the function (in this case, the trace()) to make use of it.

As you're probably already beginning to realize, you've already used arguments. When you write gotoAndStop(5)*, the* 5 *is an argument to the* gotoAndStop() *action. There's not much difference between actions and functions, except the Flash development team has already written the actions for you, and functions are something you write yourself. Fundamentally, though, they're the same thing.*

Here's the full script using arguments (it's in shopping2.fla):

```
function walkTo(destination) {
  trace("I am now walking to " + destination);
}
function findItems() {
  trace("I am finding the items...");
}
function findCost() {
  trace("I am being told the bill...");
}
function payForGoods() {
  trace("I am paying for the goods...");
}
function checkChange() {
  trace("I am checking my change...");
}
for (var i:Number = 0; i < 10; i++) {
  walkTo("the store...");
  findItems();
  walkTo("the checkout...");
  findCost();
  payForGoods();
  checkChange();
  walkTo("my home.\n\n");
}
```

The last call to the walkTo() function includes \n\n. Flash interprets \n as a newline character, so this will add two new lines after the string "my home".

Later on, if you want to make the "walking" part of this movie any more complex, you'll just need to change the declaration. You don't need to do anything to the three calls to walkTo(), since they just tell Flash to use the string passed as an argument to do whatever actions are inside the function.

Local variables and modular code

The process of fleshing out each stub into final code is best done using something programmers call a **modular process**. This means that you should be able to write each function as if it was a separate little program (or **module**) in its own right.

For this to happen, each function block should ideally have its own set of variables that are *local* to the function. For example, if two functions have a variable called myVariable, as long as they are defined as being local in scope, the two versions of myVariable will be different and separate variables—they just happen to have the same name. The advantages of doing this are massive:

- You can use the same function in different SWFs because, if each function defines its own local variables, it keeps all internal calculations to itself. The function is unaffected by other code around it.
- Because each function is effectively a self-contained mini-script, a lot of other things also stay contained within the function, the most important being *bugs*. A bug that occurs within your function will tend to make the function behave incorrectly, but it's less able to cross over into other functions and make them go wrong as well. If you use no functions at all, it's difficult to constrain errors in this way.
- You can test each function separately, which also means that you can usually design each function separately (or, at least, you can once you define your skeleton). The great thing about this is it makes it easier for a team to work on a single application. By splitting up your problem into self-contained functions, you can assign the writing of each function to a different team member. You can equally tackle each function as a separate problem yourself.

So what is a **local variable**? A local variable is one defined within a function using the keyword var. It will exist only for as long as the function is running. As soon as the function ends, the variable is deleted and freed from memory. With small scripts, this may not seem all that important, but once you start writing more complex ActionScript, memory usage becomes a major issue. Unused variables and event handlers can slow down your movies considerably. Tidying up after yourself is not only a good personal habit, it's vital when working with computer programming.

Let's see how this would be useful. Suppose you want to expand the checkChange() function to check the amount of money you'll receive. Change the checkChange() function so that it looks like this:

```
function checkChange(cost, moneyTendered, moneyReturned) {
  var money:Number = 0;
  trace("I am checking my change...");
  money = moneyTendered - cost;
  if (money == moneyReturned) {
    trace("\tCorrect change given");
  } else {
    trace("\tIncorrect amount!");
  }
}
```

The trace() commands in this function begin with \t. This tells Flash to insert a tab character in the same way that \n inserts a new line.

Using the var keyword when defining the variable money inside the function block makes the variable local in scope: it exists only within your function block and is discarded when you exit the function. It's used to give you moneyTendered - cost, and this is the value of money you expect back (the money you tender to the cashier minus the cost of your purchases). If the money you get back from the cashier, moneyReturned, is the same as the money you expect, money, all is well and good. If it isn't, you have received the incorrect amount. What you do in this case will probably depend on the level of the difference, in whose favor it is, and how indignant/honest you are, but let's leave that issue alone for now.

To see the code in action for a correct amount, remove the loop from the main code like this (the revised code is in shopping3.fla):

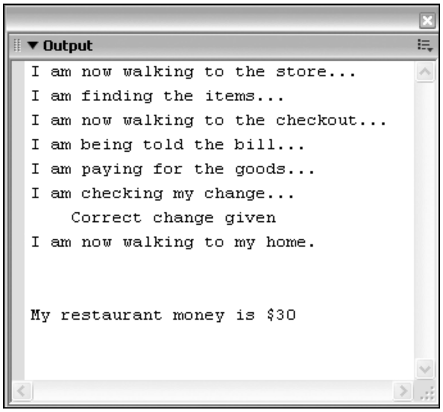
```
walkTo("the store...");
findItems();
walkTo("the checkout...");
findCost();
payForGoods();
checkChange(12.50, 15.00, 2.50);
walkTo("my home.\n\n");
```

Also, to check for an incorrect change value, try using this:

```
checkChange(12.50, 15.00, 2.55);
```

That's cool, but it doesn't really tell you the advantage of your local variable money. Well, suppose you have an amount of money to spend at a restaurant later on in the day, and you also store this in a variable called money but on the main timeline. How would this affect your money local variable in the function checkChange()? Let's find out! Add the following lines shown in bold to the code, and test the movie again (the amended code is in shopping4.fla). The result is shown alongside the code.

```
var money:Number = 30;
walkTo("the store...");
findItems();
walkTo("the checkout...");
findCost();
payForGoods();
checkChange(12.50, 15.00, 2.50);
walkTo("my home.\n\n");
trace("My restaurant money is $"+money);
```



```
I am now walking to the store...
I am finding the items...
I am now walking to the checkout...
I am being told the bill...
I am paying for the goods...
I am checking my change...
    Correct change given
I am now walking to my home.


My restaurant money is $30
```

See that? The variable money on the main timeline is unaffected by the local variable money in the function checkChange(). Here's the sequence of events that makes this possible:

1. At the start of the main code, you define a variable called money.
2. You run through the functions. One of them, checkChange(), creates a different variable also called money within its code block. This version of money is constrained to the function block in which it was created, so it doesn't affect the version on the main timeline.
3. When you check the money value on the main timeline, it's still $30.00, so you can go and have that meal after all.

Now try it again, but this time temporarily disable the line that starts with var in the checkChange() function by typing two forward slashes at the beginning of the line. (This turns it into a comment, so that it's ignored when the script is run—a useful technique for testing the impact of individual lines on a script.) This time, the variable money *is* changed. Why?

Well, when within a function, Flash will assume a local variable *only if there's one defined* using the var keyword. If there isn't a local variable called money defined, the function will be able to see the one outside the function and will use that.

> *This is a consequence of scope and the **scope chain**. A variable defined within a function using* var *will "mask out" a same-named variable defined outside the function. If there's no local variable, the version used will be the one outside the function, but on the same timeline. If there's no variable on the current timeline with the same name, then Flash will look in* _global*. (The use of* _global *is covered in Chapter 8.) If Flash still can't find a variable of the required name, it will either return* undefined *(if you're trying to read the variable value) or create a variable on the current timeline (if you're trying to write a variable value).*

Returning values from a function

Let's look at another function stub and refine it a little. Looking at the train of events in the shopping trip, you could reasonably expect that the findCost() function would be required to calculate the bill by adding up the prices of every item you bought. There are two problems here.

- You don't know how many items you've bought. As well as the prices for the items being different, the number of items is different every time, so you don't know how many arguments to add. The way around this is easy: use an array. An array is essentially a list of values, which is just what you need here.
- You haven't defined what you've bought anywhere yet. That's no problem either—you can simply make up some data for now. Making up some variables (or code) so that your functions under development can be tested properly (even though your skeleton isn't yet developed enough to work with the functions) is called adding a **test harness**. A test harness is a bit of code you add so that you can develop a module (function) further without having to wait for other code (which is required to provide the real data) to be written.

OK, test harness first. Make money a local variable again inside the checkChange() function by removing the double slashes you added in the previous section. Then amend the main code like this (new code is shown in bold):

```
var money:Number = 30;
var priceTags:Array = new Array(0.50, 1.40, 3.60, 7.00);
walkTo("the store...");
findItems();
walkTo("the checkout...");
findCost();
payForGoods();
checkChange(12.50, 15.00, 2.50);
walkTo("my home.\n\n");
trace("My restaurant money is $"+money);
```

You've defined an array of price tags called priceTags. This array contains a set of numbers to represent the goods bought: $0.50, $1.40, $3.60, and $7.00. If you were developing a real application, these values would be generated by findItems(), but for the purposes of this exercise, we've just had you insert the prices directly into the code as an array.

Next up, you need to write the findCost() function code. Change the findCost() function as shown here:

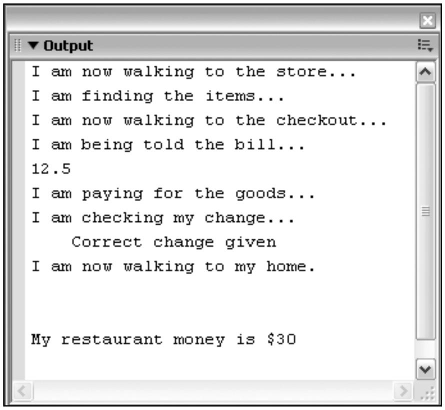
```
function findCost(prices) {
  var money:Number = 0;
  for (var i:Number = 0; i < prices.length; i++) {
    money += prices[i];
  }
  trace("I am being told the bill...");
  return money;
}
```

The function takes an argument, prices, and uses a local variable, money (so now you have *three* versions of the same variable!), to store the sum of each element in prices. (In real life, you would need to check that prices is an array, and that each element contains a number, but we're trying to keep things simple for the moment.) The last line of this function is something you haven't seen before: a return. A returned value is just that—a value that's returned to the main script. You can store this returned value by assigning the function call to a variable like this:

```
var myVariable:Number = findCost(prices);
```

This will return the value of money (the version in findCost() that is equal to the sum of all the prices) and store it in myVariable. Try it by changing the main code to start using this new function (or use shopping5.fla):

```
var money:Number = 30;
var priceTags:Array = new Array(0.50, 1.40, 3.60, 7.00);
var total:Number = 0;
walkTo("the store...");
findItems();
walkTo("the checkout...");
total = findCost(priceTags);
trace(total);
payForGoods();
checkChange(12.50, 15.00, 2.50);
walkTo("my home.\n\n");
trace("My restaurant money is $"+money);
```



```
I am now walking to the store...
I am finding the items...
I am now walking to the checkout...
I am being told the bill...
12.5
I am paying for the goods...
I am checking my change...
    Correct change given
I am now walking to my home.


My restaurant money is $30
```

When you trace the value total, you see that it has taken on the value of the sum of your prices, courtesy of the code in findCost(). As you can see from the screenshot, you would need to add some extra code to format the price correctly as currency, but this shows the basic principle of how a function returns a value.

Typing functions

So far, you've been making lots of small changes to the code, and looking at it now you can see already something is beginning to emerge that's starting to look seriously complicated. Luckily, this complexity is being managed by your use of functions. You look at only one thing at a time by working on one function at a time. In short, the problem is split into distinct bits of manageable code via functions.

The next thing you'll examine will make your functions look like that geeky hard-core stuff you see in big, thick books you'd rather not read, but trust me, it's really not that bad. In fact, it's designed to make your life easier.

In the same way you can type variables, you can also type functions. There are two things you can type: the arguments and the returned value. The returned value can be either nothing or something. If it's something, it has to be one of the following types: `Number`, `String`, `Array`, or `Boolean`. If it's nothing, Flash uses `Void`, which is computer-speak for nothing.

Confusingly, ActionScript also has an operator called `void` (all lowercase), which is so rarely used, you can forget about it. Just remember that all the types used in variable and function checking begin with an initial capital. If you use all lowercase, your movie will fail to compile.

A function can have a type defined for each argument, plus one for the returned value:

```
function myFunction(argument:type, argument:type):type {
  // do stuff
}
```

For each argument, you can define a type in much the same way as you do a variable. For the return value, you place the type after a colon between) and { at the end of the first line of the function definition. If the function doesn't return any value, then the return type is `Void`.

The listing with fully typed functions looks like this (you can also find it in `shopping_typed.fla`):

```
function walkTo(destination:String):Void {
  trace("I am now walking to " + destination);
}
function findItems():Void {
  trace("I am finding the items...");
}
function findCost(prices:Array):Number {
  var money:Number = 0;
  for (var i:Number = 0; i < prices.length; i++) {
    money += prices[i];
  }
  trace("I am being told the bill...");
  return money;
}
function payForGoods():Void {
  trace("I am paying for the goods...");
}
```

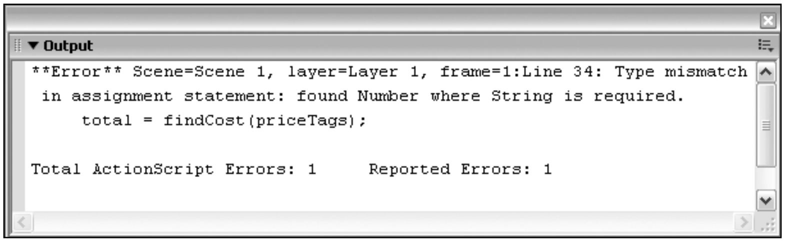
```
function checkChange(cost:Number, moneyTendered:Number, ➥
moneyReturned:Number):Void {
  var money:Number = 0;
  trace("I am checking my change...");
  money = moneyTendered - cost;
  if (money == moneyReturned) {
    trace("\tCorrect change given");
  } else {
    trace("\tIncorrect amount!");
  }
}
var money:Number = 30;
var priceTags:Array = new Array(0.50, 1.40, 3.60, 7.00);
var total:Number = 0;
walkTo("the store...");
findItems();
walkTo("the checkout...");
total = findCost(priceTags);
trace(total);
payForGoods();
checkChange(12.50, 15.00, 2.50);
walkTo("my home.\n\n");
trace("My restaurant money is $"+money);
```

The advantage of using typing in this way is that it tightens up your skeleton. If you use Void for a function where you actually return a value, Flash will raise an error. If you set an argument as a String but then make that argument a Number, Flash will raise an error. The following screenshot shows what happened when this line:

```
var total:Number = 0;
```

was altered like this:

```
var total:String = "0";
```



```
**Error** Scene=Scene 1, layer=Layer 1, frame=1:Line 34: Type mismatch
 in assignment statement: found Number where String is required.
     total = findCost(priceTags);

Total ActionScript Errors: 1     Reported Errors: 1
```

The error message not only tells you that the compiler found a number where it expected to find a string, it pinpoints exactly where the mistake occurred (line 34). The compiler is telling you that findCost() returns a number, so total must be of the same data type—it cannot be a string. In this case, the error was induced deliberately, but in more complex scripts, it's easy to lose track of changes that you have made. Using a string where a number is expected may produce unexpected results, the cause of which can be very difficult to track down. So, although typing makes Flash very fussy about what it will allow to run, it's actually a very good thing. Although it takes more effort to make code run, when it does, *the code is more likely to run correctly*. Typing tends to create more robust and structured code than untyped styles.

Typing functions makes your skeleton much more precise in the way it defines the flow of data between the main code and functions (and, in more complex code, between functions). It prevents you from making big mistakes later on in the development cycle by imposing some well-defined skeleton code early in the planning stage.

Typing is very useful, but it really comes into its own in longer scripts. You'll use typing extensively in the Futuremedia site code later on in the book.

Running in circles

Say we wanted to give you a real scare. One of us could put a sheet over his head and jump out at you shouting "Boo!" On the other hand, we could just whisper the word "trigonometry" in your ear. Similar effect, probably.

Don't panic—we're not about to try to give you a boring lesson in pure math. We're just going to show you two very cool little functions that you and your movies might find rather useful.

```
function circleX(distance:Number, angle:Number):Number {
  return distance * Math.sin(Math.PI * angle/6);
}
function circleY(distance:Number, angle:Number):Number {
  return -distance * Math.cos(Math.PI * angle/6);
}
```

Now, if you can look at these lines without breaking out in a cold sweat, that's great. If not, it doesn't matter, because we're not going to talk about *how* these functions work, and you don't need to understand any of what's going on inside them. All that's really important is *what* they do.

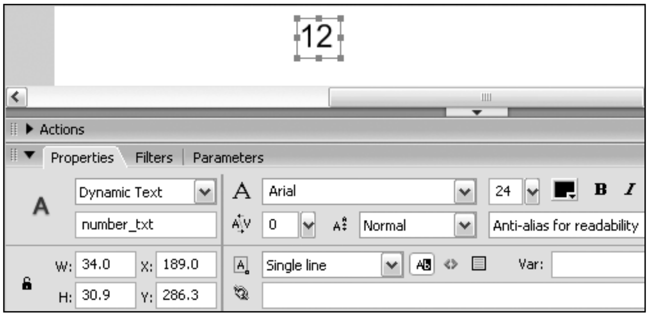
So far in this book, all the ActionScript animation you've looked at has dealt with things moving about the stage in straight lines. Even the moving blobs you created in Chapter 5 were basically just an animated straight line connecting the pointer to a fixed point on the stage.

That's pretty cool as far as it goes, but what about making circles instead of straight lines? Sounds hard, but with these two functions, it's not a problem. You can use the functions to turn circle-based values like "distance from center" and "angle" (that is, how far around the circle you are) into x- and y-coordinates that you can use to position things on the stage.

Building the clock face

Let's start off by drawing a clock face—well, the numbers on a clock face anyway. The final code for this exercise is in `clock1.fla`.

1. Create a new Flash document, change the frame rate to 20 fps, and create a dynamic text field on the stage. Make sure the field is big enough to hold the text 12 in whichever type you decide to use. Enter number_txt in the instance name field of the Property inspector.



12

Actions

Properties | Filters | Parameters

Dynamic Text

number_txt

Arial 24

0 Normal Anti-alias for readability

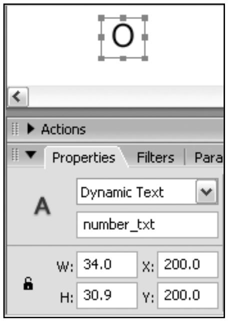
W: 34.0 X: 189.0

H: 30.9 Y: 286.3

Single line

Var:

2. Use the Property inspector to place the text field at X: 200, Y: 200; replace 12 with the uppercase letter O; and click the Align Center button (at the top right of the Property inspector) to center the text.



O

Actions

Properties | Filters | Para

Dynamic Text

number_txt

W: 34.0 X: 200.0

H: 30.9 Y: 200.0

> *Using x- and y-coordinates of 200 means the clock won't be in the exact center of the stage, but that's not important for this exercise. Using round figures makes the calculations easier, allowing you to concentrate on what's happening, rather than fiddling around with a lot of odd numbers.*

3. Now add a new layer called actions, open the Actions panel, and click the pushpin button to make sure you don't accidentally lose focus. Let's start by putting in the two functions we showed you just a few moments ago. Just key them in exactly as shown:

```
function circleX(distance:Number, angle:Number):Number {
  return distance * Math.sin(Math.PI * angle/6);
}
function circleY(distance:Number, angle:Number):Number {
  return -distance * Math.cos(Math.PI * angle/6);
}
```

4. Now for the main script. It's very simple—you'll just loop through values 1 to 12 and make a duplicate of the number_txt text field each time:

```
for (var i:Number = 1; i < 13; i++) {
  duplicateMovieClip(number_txt, "number"+i+"_txt", i);
  // position text field
  // change text in text field
}
```

5. As you can see, we've added comments to mark where you need to add a couple more things. Let's start with the second of these, since it's the simpler of the two:

```
for (var i:Number = 1; i < 13; i++) {
  duplicateMovieClip(number_txt, "number"+i+"_txt", i);
  // position text field
  // change text in text field
  _root["number"+i+"_txt"].text = i;
}
```

This is a new way of setting the text in a text field, but hopefully it shouldn't worry you too much. In Chapter 3, you used the Var field in the Property inspector to give Flash the name of a variable to store the text in. You don't do that this time. Since you're making 12 copies of the text field, they'd all end up using the *same* variable! Instead, you hack straight into the dynamic text field's text property—just as you've done in the past with other properties such as _x, _y, and _visible—and change it to the current value of i.

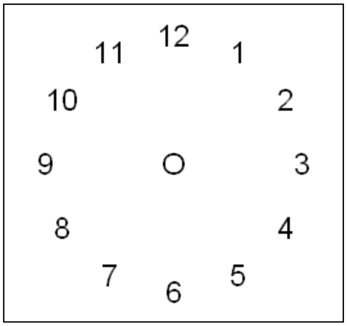
This gives you 12 new text fields, showing numbers 1 through 12. Note that the for loop starts at i=1 and stops as soon as i hits 13. Run the movie now, and you'll see all the text fields sitting on top of each other, as shown alongside. If you want to check the mess for yourself, right-click/Ctrl-click and zoom into the movie to see the gory details.

Ugh! Let's move them, quick!

6. The plan is to put the 12 new text fields in a circle around the original one, which still has its top-left corner sitting at X:200, Y:200. This is where your functions come in. You give them two arguments: a distance from the center and the number of a position around the edge of the circle. Here's the script you need to add:

```
var center:Number = 200;
for (var i:Number = 1; i < 13; i++) {
  duplicateMovieClip(number_txt, "number"+i+"_txt", i);
  // position text field
  _root["number"+i+"_txt"]._x = center + circleX(100, i);
  _root["number"+i+"_txt"]._y = center + circleY(100, i);
  // change text in text field
  _root["number"+i+"_txt"].text = i;
}
```

For each text field, you set the x-coordinate to 200 (stored in the variable center) plus the return value of circleX() and the y-coordinate to 200 (again derived from the variable center) plus the return value of circleY(). You'll look at how you call these functions in a moment. First, though, reassure yourself that this does actually work by testing the movie now. This is what you should see:



Cool! You've made a circle without resorting to any evil mathematical equations. Well, actually you have, but they're hidden away inside the functions, so you don't need to worry about them. All that really matters is this pair of lines:

```
_root["number"+i+"_txt"]._x = center + circleX(100, i);
_root["number"+i+"_txt"]._y = center + circleY(100, i);
```

and the fact that circleX() gives you the x-coordinate for each number, while circleY() gives you the y-coordinate for each number. What about all those numbers you used on the right side, though? They do look a bit confusing, so let's figure out what they do.

Changing how the circle appears

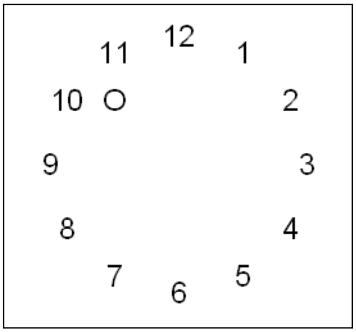
All the values in the first line determine x-coordinates for the text fields, while all those in the second line control y-coordinates for the text fields. There are three different things to look at in each line, and each controls a different aspect of how the overall circle appears.

1. First up, each line's right side begins with center. This tells Flash what point on the stage to draw the circle around. Try changing the value of center to see how it affects things:

   ```
   var center:Number = 250;
   ```

   Now you should see something like this (the code is in clock2.fla):

   The central O is still where it was, but all the numbers have moved down and across to the right because you've added 50 to the center coordinates.



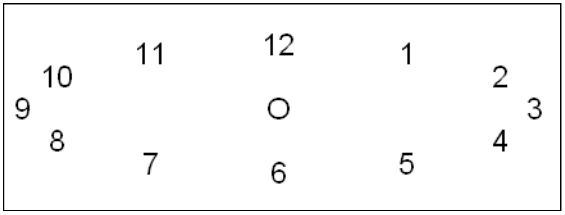
2. OK, so the first pair of values controls the position of the circle. What about the first of the arguments in each of the functions? At the moment, they're both set to 100, so let's try changing them and see what happens. First reset the value of center to 200:

   ```
   var center:Number = 200;
   ```

   Then change the first arguments inside circleX() and circleY(), like this (or use clock3.fla):

   ```
   _root["number"+i+"_txt"]._x = center + circleX(200, i);
   _root["number"+i+"_txt"]._y = center + circleY(50, i);
   ```

You doubled the top value and halved the bottom one, so you should now see something like this:

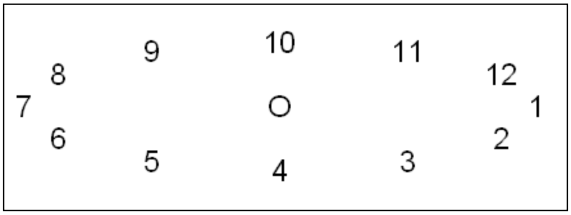


Yes, you can use the first argument in `circleX()` to control how wide the "circle" is and the first argument in `circleY()` to control how tall it is. In fact, the values tell Flash the horizontal and vertical distances from center to edge, so this "circle" is all of 400 pixels wide and 100 pixels high.

3. Now for the second of the arguments. Right now this is simply `i` for both of them, and you can probably guess that this is what controls the angle (or rotation) of each position—that is, where on the circle each number will appear. If the second argument has a value of 6, you'll get a position at the bottom of the circle. Likewise, a value of 9 gives a position on the far left of the circle. To see this in effect, let's play with the values again (the code is in `clock4.fla`):

```
_root["number"+i+"_txt"]._x = center + circleX(200, i + 2);
_root["number"+i+"_txt"]._y = center + circleY(50, i + 2);
```

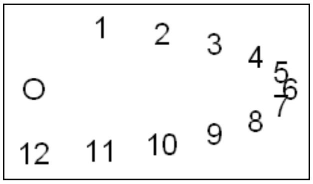
This will give you the following:



All the numbers have shifted around the clock face by two places. Even better, you don't need to stick to using whole numbers—for example, a value of 1.5 will give you a position halfway between the 1 and 2 positions. So what about halving the value of `i`?

```
_root["number"+i+"_txt"]._x = 200+circleX(200, i/2);
_root["number"+i+"_txt"]._y = 200+circleY(50, i/2);
```

Now the numbers will run from clock positions 0.5 to 6, so they're all on the right side of the circle, and none remain on the left (see `clock5.fla`).



As you can see, with these two circle functions, you can do some pretty impressive stuff. What's more, you don't need to know anything about what goes on inside them—you can just plug in a few numbers and draw circles, ovals, and arcs to your heart's content! If you look at clock_spiral.fla in the download files for this chapter, you'll see just one example of the nonsense you can get up to with a few calculations.

Before you get too carried away with this, there's another important lesson to learn about functions and what you can do with them.

Nesting functions

It may seem like an obvious point to make, but there's no reason you can't call one function from inside another. Say you have a simple function, double(), declared like this:

```
function double(x:Number):Number {
  x *= 2;
  return x;
}
```

Now, if you want to create a new function quadruple() that multiplies a specified value by 4, you can set it up so that it calls the double() function twice over:

```
function quadruple(y:Number):Number {
  y = double(y);
  y = double(y);
  return y;
}
```

In fact, you could make this even shorter by actually passing the first double() function call as an argument in the second:

```
function quadruple(y:Number):Number {
  y = double(double(y));
  return y;
}
```

Here, Flash first calculates the result of the second function call, and then it uses this value as the argument for the first function call. Now, if you follow it up with something like this:

```
var x:Number = 4;
x = quadruple(x);
trace(x);
```

you'll get the value 16 in the Output window. First Flash doubles 4 to get 8, and then it doubles 8 to get 16. Easy. But what does this mean for scriptwriting in general?

It means that you can have as many layers of functions as you want between your overall problem and the separate actions that make up a scripted solution. Not only can you break down the task into various subtasks, but also you can break down each of the subtasks into its own sub-subtasks, and so on, for as long as you find it useful.

Using nested functions

The most obvious benefit of nesting functions is that you can call your own functions from inside an event handler. For instance, if you put your clock face script inside the onEnterFrame event handler for the root timeline, you can update it in real time.

Chasing the mouse

Let's have some fun with this right away and make the number circle follow your mouse pointer around the stage.

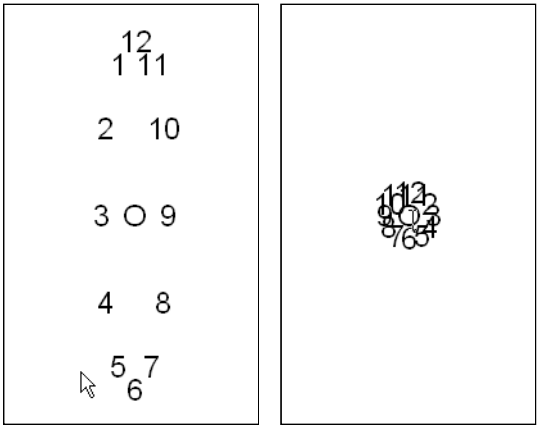
1. Put the main script in an onEnterFrame event handler (so it updates all the text field positions once every frame), and change center to _xmouse and _ymouse, like this (the code is in clock6.fla):

```
function circleX(distance:Number, angle:Number):Number {
  return distance * Math.sin(Math.PI * angle/6);
}
function circleY(distance:Number, angle:Number):Number {
  return -distance * Math.cos(Math.PI * angle/6);
}
var center:Number = 200;
onEnterFrame = function() {
  for (var i:Number = 1; i < 13; i++) {
    duplicateMovieClip(number_txt, "number"+i+"_txt", i);
    // position text field
    _root["number"+i+"_txt"]._x = _xmouse + circleX(100, i);
    _root["number"+i+"_txt"]._y = _ymouse + circleY(100, i);
    // change text in text field
    _root["number"+i+"_txt"].text = i;
  }
};
```

2. Run the movie again, and sure enough, the circle of numbers goes where the mouse pointer leads it. Wherever the pointer is, it's always in the middle of the circle.
3. Now let's see what happens if you use the mouse pointer to control the circle's width and height. Don't forget to put the center variable back in, or you'll see some very strange things going on!

```
_root["number"+i+"_txt"]._x = center + circleX(_xmouse-200, i);
_root["number"+i+"_txt"]._y = center + circleY(_ymouse-200, i);
```

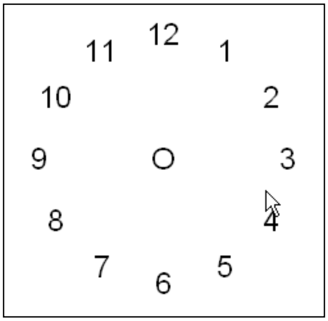
Now, as you move the mouse pointer across the stage, the circle's width changes accordingly. Likewise, the circle's height changes as you move the mouse pointer up and down the stage. (You can check the code in clock7.fla.)



The values of _xmouse-200 and _ymouse-200 collapse the circle when the mouse pointer's directly over the O in the center of the clock (that is, when both have values of 200).

4. No, it isn't looking much like a circle any more, is it? Let's use the same value to control both the width *and* the height. You'll get the value from the x-coordinate of the mouse pointer and store it in a variable called size, like this (see clock8.fla):

```
var size:Number = _xmouse - 200;
_root["number"+i+"_txt"]._x = center + circleX(size, i);
_root["number"+i+"_txt"]._y = center + circleY(size, i);
```



5. OK, you've figured out how to tweak the position of the circle, and you've just been fiddling with the width and height. What else is there? Well, there's still another function argument to play with.

   Let's use _ymouse to control the rotation of the number circle. Since _ymouse can take values up to 500 or so, you'll divide it by 100 first, which will give you lots of fine control but with a rotation range of about 5. That's almost halfway around the clock, which should be enough for you to see that it's working OK.

   ```
   var size:Number = _xmouse - 200;
   var rotation:Number = i + _ymouse/100;
   _root["number"+i+"_txt"]._x = center + circleX(size, rotation);
   _root["number"+i+"_txt"]._y = center + circleY(size, rotation);
   ```

   Of course, you still need to add i to the _ymouse-based value; otherwise, all the text fields will end up in the same place. Now the circle turns around as you move the mouse up and down.

6. Let's quickly recap the full script listing (it's also in clock9.fla):

   ```
   function circleX(distance:Number, angle:Number):Number {
     return distance * Math.sin(Math.PI * angle/6);
   }
   function circleY(distance:Number, angle:Number):Number {
     return -distance * Math.cos(Math.PI * angle/6);
   }
   var center:Number = 200;
   onEnterFrame = function () {
     for (var i:Number = 1; i < 13; i++) {
       duplicateMovieClip(number_txt, "number"+i+"_txt", i);
       // position text field
       var size:Number = _xmouse - 200;
       var rotation:Number = i + _ymouse/100;
       _root["number"+i+"_txt"]._x = center + circleX(size, rotation);
       _root["number"+i+"_txt"]._y = center + circleY(size, rotation);
       // change text in text field
       _root["number"+i+"_txt"].text = i;
     }
   };
   ```

   Not bad, considering what you've achieved!

Using more function nesting to tidy up your script

Before we wrap up this series of experiments, we'll show you how to tighten up the script a little. Not everything in the onEnterFrame event handler actually needs to be there, and it makes sense to put the lines of code that position the text fields into their own function (so you can turn variables like size and rotation into function parameters and make the code easier to customize in future).

Let's leave the opening pair of functions alone (unless you're feeling very adventurous) and brew up a couple of brand-new ones just underneath.

1. The first of these new functions is called createTextFields(). It creates all the text fields for you when the movie starts.

   ```
   function createTextFields():Void {
     for (var i:Number = 1; i < 13; i++) {
       duplicateMovieClip(number_txt, "number"+i+"_txt", i);
       // change text in text field
       _root["number"+i+"_txt"].text = i;
     }
   }
   ```

   You're assuming that the actual text you put into each of the fields isn't going to change while the movie's running, so you set that up here as well.

2. You'll put all the other stuff into a function called positionTextFields().

   ```
   function positionTextFields(centerX:Number, centerY:Number, ➥
   size:Number, rotation:Number):Void {
     for (var i:Number = 1; i < 13; i++) {
       var field:String = "number"+i+"_txt";
       _root[field]._x = centerX + circleX(size, i + rotation);
       _root[field]._y = centerY + circleY(size, i + rotation);
     }
   }
   ```

   This function has four parameters that control exactly where the circle appears, how large the circle is, and the angle at which the numbers start around the circle. The variable field simply stores the name of the text field to make it easier to access in the following two lines. It's also marginally more efficient, because ActionScript needs to calculate the value only once, rather than on each reference.

3. Now that all the positioning script is tidily wrapped inside a function, the onEnterFrame event handler gets a lot simpler:

   ```
   onEnterFrame = function() {
       positionTextFields(200, 200, _xmouse-200, _ymouse/100);
   };
   ```

4. Finally, you need to run the createTextFields() action, so that the fields actually get set up:

   ```
   createTextFields();
   ```

   You can leave everything else to the onEnterFrame handler function.

5. Here's the finished script (you can also find it in clock_final.fla):

   ```
   // DECLARE FUNCTIONS FIRST
   function circleX(distance:Number, angle:Number):Number {
     return distance * Math.sin(Math.PI * angle/6);
   }
   function circleY(distance:Number, angle:Number):Number {
     return -distance * Math.cos(Math.PI * angle/6);
   }
   ```

```
function createTextFields():Void {
  for (var i:Number = 1; i < 13; i++) {
    duplicateMovieClip(number_txt, "number"+i+"_txt", i);
    // change text in text field
    _root["number"+i+"_txt"].text = i;
  }
}
function positionTextFields(centerX:Number, centerY:Number, ➥
size:Number, rotation:Number):Void {
  for (var i:Number = 1; i < 13; i++) {
    var field:String = "number"+i+"_txt";
    _root[field]._x = centerX + circleX(size, i + rotation);
    _root[field]._y = centerY + circleY(size, i + rotation);
  }
}
// DECLARE onEnterFrame EVENT HANDLER
onEnterFrame = function() {
    positionTextFields(200, 200, _xmouse-200, _ymouse/100);
};
// ACTIONS TO RUN WHEN MOVIE STARTS
createTextFields();
```

Hmm. The code is actually a bit *longer* than it was before. So why is it worth making all those changes? Well, apart from the slight improvement in speed (now that you're not making a whole set of duplicate text fields every frame), you've pulled out the main circle variables as parameters in the positionTextFields() function. This makes it loads easier to change how you control the appearance of the circle. Of course, you've hard-wired certain features into the function—for example, the width and height will always be the same, the numbers will always be evenly spaced around the whole circle, and so on. There's nothing stopping you from changing that, though.

Now that you've seen how easy it is to use functions like circleX() and circleY() to make a funky little circle of text fields, you might like to start experimenting with them yourself. What you've done here is only the tip of the iceberg, and there are plenty of other possibilities left for you to explore. What about changing the value of rotation steadily from one frame to the next? Or using different values of size for the different text fields, so they spiral into the center? Or even changing the text fields for something completely different, like an animated movie clip? Now you have the basics down, the sky is the limit!

The important thing to realize in all this playing around is that you've taken a general feature (that is, being able to define points around a circle) and written it as a function. You've then largely forgotten about the circleX() and circleY() functions, and played around with the script that uses the functions—you've been able to forget the code inside the functions.

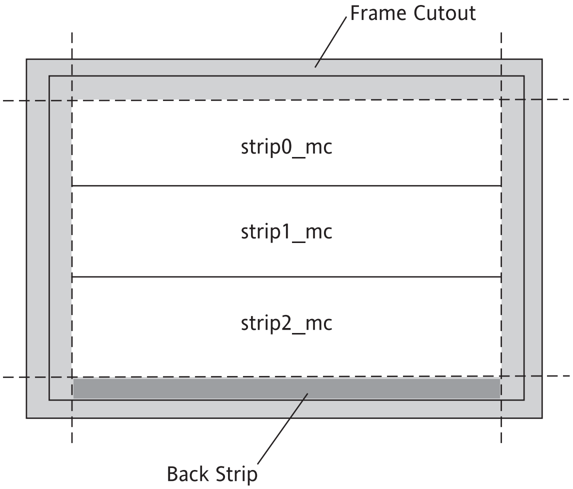
This process of building a function, making it work, then forgetting about how it works and just using it, is a hallmark of modular design in general. Not only does it prevent you from constantly reinventing the wheel, but also it allows you to repurpose code to do several things or simply to experiment.

Book project: Creating the dynamic graphics

You must be wondering by now if you're ever going to get around to scripting the Futuremedia site. You will in the next chapter, we promise you, but first you need to build the graphics that will be controlled by the ActionScript. It's time to build the dynamic part of the interface.

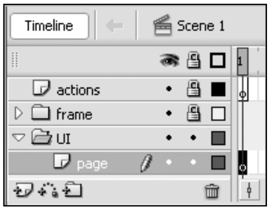
What's missing are the three colored strips in the middle of the stage, together with the little colored icons that form a breadcrumb trail across the bottom of the page. Be warned, though, that for the time being everything will be the same boring battleship gray that you used when building the mockup in Chapter 2.

Before you begin, you need to do a little calculation. As the following diagram shows, the three strips (called rather imaginatively `strip0_mc`, `strip1_mc`, and `strip2_mc`) fit inside the frame cutout that you created in the last chapter. If you remember, it was decided to make the stage 800 × 600 pixels with a 30-pixel border around each side. That leaves a space of 740 × 540 pixels for the strips. If you divide 540 by 3, it comes to 180. So, each strip needs to be 740 pixels wide and 180 pixels high. Let's get to work! (The final FLA for this section is `futuremedia_setup3.fla`.)

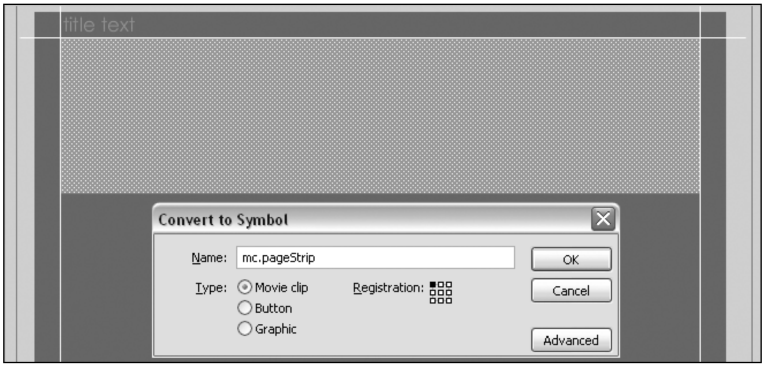


Making the strip movie clips

1. Continue working with the FLA from the last chapter or open `futuremedia_setup2.fla`. To avoid adding anything to the frame layer folder by mistake, click the triangle to the side of the folder name to close it, and then lock the folder. Also lock the actions layer, if you haven't already done so.
2. Inside the UI layer folder, add a new layer called page. Your timeline should now look like the screenshot shown alongside.



3. Select the Rectangle tool with fill color #999999 and no stroke. Draw a rectangle on the page layer in any free area.
4. Select the rectangle and use the Property inspector to resize it to exactly 740 × 180 pixels. Set both the X and Y coordinates to 30. The rectangle should now fit snugly between the guidelines in the top section of the frame.
5. You need to turn the rectangle into a movie clip, so press F8. Call it mc.pageStrip and set the registration point to the top left, as shown in the following screenshot.



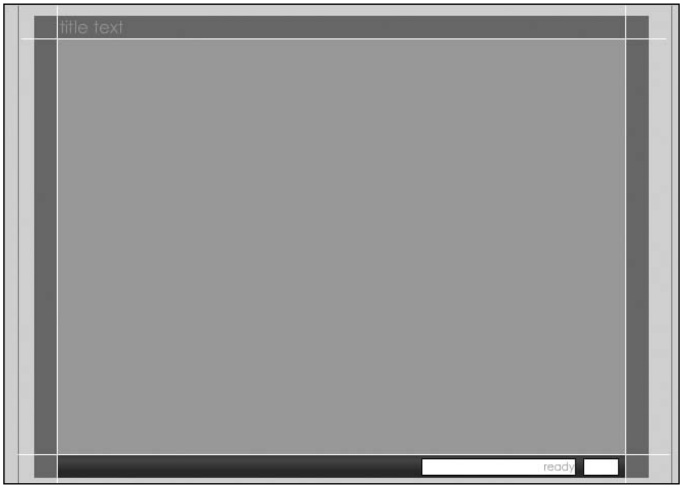
Name: mc.pageStrip

Type: ● Movie clip ○ Button ○ Graphic

Registration: top left

As explained in Chapter 5, Flash stores symbols in the Library panel in alphabetical order, so adding mc. to the front of a movie clip name makes it possible to keep all movie clips together, in the same way as adding sy. to the front of graphic symbols keeps all of them together.

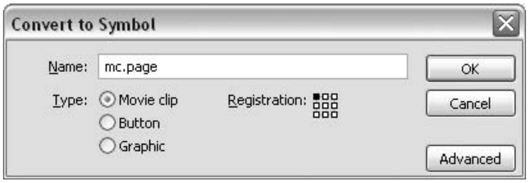
6. Give the movie clip an instance name of strip0_mc. Drag two more instances of mc.pageStrip from the Library panel. Position the first one at X: 30, Y: 210, and give it an instance name of strip1_mc. The second one should go at X: 30, Y: 390, and be named strip2_mc.
7. The central area of the stage should now be completely filled with light gray, as shown in the following screenshot, and there should be no gaps between any of the strips.



With all of the `mc.pageStrip` instances deselected, you should see what appears to be one solid rectangle. If gaps occur between any of the `mc.pageStrip` instances, go back and check your measurements. The successful display of the page navigation depends on a pixel-perfect fit at this stage.

Forming the tricolor movie clip

1. Select all three strips (`strip0_mc`, `strip1_mc`, and `strip2_mc`) by holding down the Shift key and clicking each one in turn. Make sure you don't accidentally move any of the strips in the process.
2. Turn the selected strips into a single movie clip by pressing F8. Call the new movie clip mc.page, and set the registration point to the top left, as shown. This last step is vitally important, as all the ActionScript measurements will be made from the top left of the new movie clip.



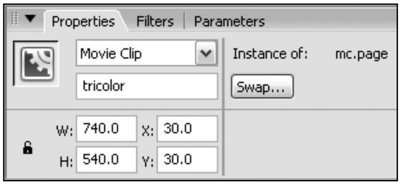
Convert to Symbol

Name: mc.page

Type: ● Movie clip ○ Button ○ Graphic

Registration: top left

3. Give the new movie clip an instance name of tricolor. Although it's still a dull gray, that will all change once the ActionScript magic gets to work. The movie clip's dimensions in the Property inspector should look like those in the screenshot alongside.



Movie Clip

tricolor

Instance of: mc.page

W: 740.0 X: 30.0

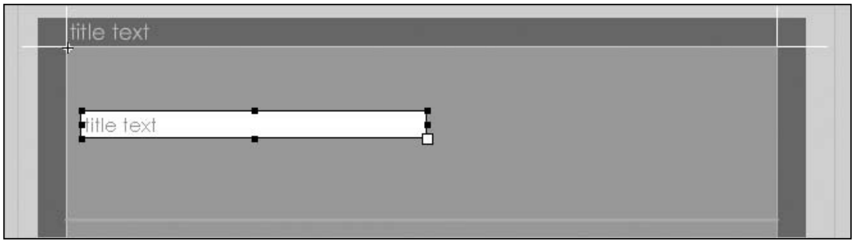
H: 540.0 Y: 30.0

4. Double-click the `tricolor` instance of the `mc.page` movie clip to edit it in place.
5. Rename the existing layer to back, and add two new layers above it called text and guide. Lock the back and text layers.
6. In layer guide, add some construction lines to show that the big rectangle is really three smaller clips. Make the lines about 3 pixels thick and use a bright color, such as green, to contrast with the grays. Position them at y-coordinates 180 and 360.
7. Make the layer a guide layer by right-clicking/Ctrl-clicking the layer title and selecting Guide from the context menu.
8. Lock the layer guide. You'll add some text next, so unlock the text layer and continue with the next section.

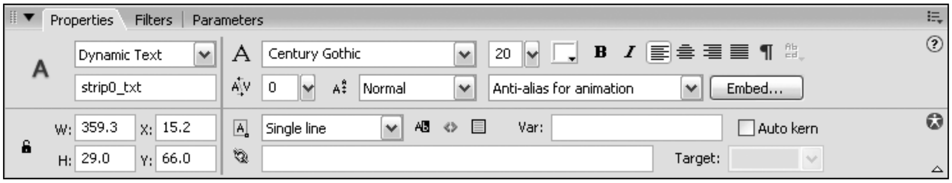
Adding the strip titles

Each strip needs a dynamic text field that will display the title of the section that it links to. Although the text will be controlled dynamically by ActionScript, the position of the text field will remain constant, so it can be placed by eye.

1. Place a dynamic text field on `strip0_mc.` Set the font to Century Gothic, font size to 20 points, and color to white. Type some dummy text in the field, and use the square handle at the bottom-right corner to drag the text field to roughly halfway across the strip. This should give you plenty of space in case any of your page titles are longer than one or two words. It's important to use the square handle to resize the field. Otherwise, your text will be distorted.

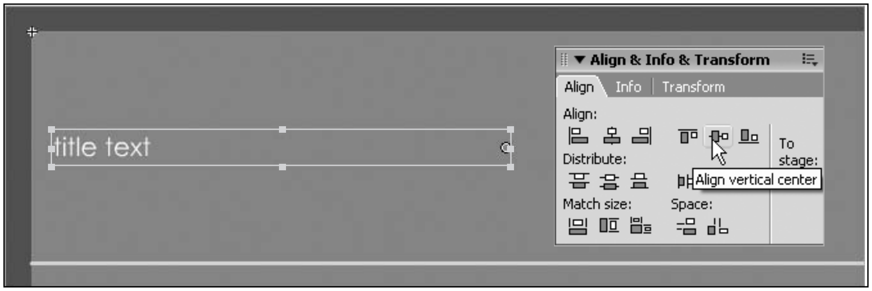


2. Give the text field an instance name of strip0_txt. The values in the Property inspector should look like the following screenshot, although the exact size and position of the text field are not important as long as the position looks good to you.


Properties | Filters | Parameters

Dynamic Text
strip0_txt
Century Gothic
20
0
Normal
Anti-alias for animation
Embed...
W: 359.3 X: 15.2
H: 29.0 Y: 66.0
Single line
Var:
□ Auto kern
Target:

However, if you want to be pixel-perfect, unlock the back layer and select `strip0_mc`. Hold down the Shift key and select `strip0_txt`. With both selected, open the Align panel (Window ➤ Align), make sure that the To stage icon is deselected, and click Align vertical center. Make sure that only the text field moves, and not the strip. If you use this technique, click in an empty part of the workspace to deselect both objects, and then select the dynamic text field on its own before going on to the next step.

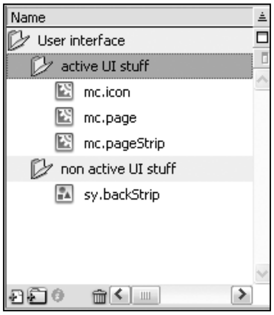


3. Click the Embed button in the Property inspector, and select Lowercase [a..z] (27 glyphs) in the Character Embedding dialog box. Click OK. This will embed the correct font outlines for the characters.
4. Make sure the text field is still selected. Then choose Edit ➤ Copy followed immediately by Edit ➤ Paste in Place (or if you're a keyboard shortcut whiz, press Ctrl+C/Cmd+C, followed by Ctrl+Shift+V/Shift+Cmd+V) to copy the text field and paste it in place. This creates a duplicate directly on top of the existing text field. Use your keyboard down arrow to move the duplicate down to `strip1_mc`. (If you keep the Shift key pressed down, it will move 10 pixels at a time. When it's close to the position you want, release the Shift key and maneuver the field into position with the up and down keys, or use the Align panel as described in step 2.) Change the instance name to strip1_txt.
5. Repeat step 4 to position a dynamic text field on `strip2_mc`. Give the text field an instance name of strip2_txt. By using this technique, all three text fields should be in perfect alignment.
6. That completes the movie clip needed for the main pages. Click Scene 1 in the Timeline panel to exit the edit-in-place screen.

All that remains is to create the little icons that form a breadcrumb trail across the bottom of the page. These are simply smaller versions of the main pages, and they work in much the same way. In fact, they use almost exactly the same movie clip.

Creating the breadcrumb icon

1. In the Library panel, highlight the `mc.page` movie clip symbol, right-click/Ctrl-click, and select Duplicate from the context menu. In the Duplicate Symbol dialog box, enter mc.icon in the Name field and click OK.
2. Double-click the new movie clip symbol to edit it. Delete layer text.
3. Unlock layer guide, right-click/Ctrl-click the layer title, and select Guide in the context menu to turn it back into a normal layer.
4. Select both horizontal lines on layer guide, and use the Property inspector to set the stroke width to Hairline and the color to black (#000000) with 40% alpha.
5. Click Scene 1 in the Timeline panel to exit the edit screen.
6. Finally, tidy up the Library panel by adding a new folder called `active UI stuff` inside the `User interface` folder. Drag the three movie clips you have just created inside, as shown, and save your FLA. If you want to check that you have everything right, compare it with `futuremedia_setup3.fla` in the download files.



Summary

Functions give you a way to bundle useful sets of actions so that you can define them once and reuse them as many times as you like. Not only can this help cut down on repetition, but also it can help you make your scripts more structured.

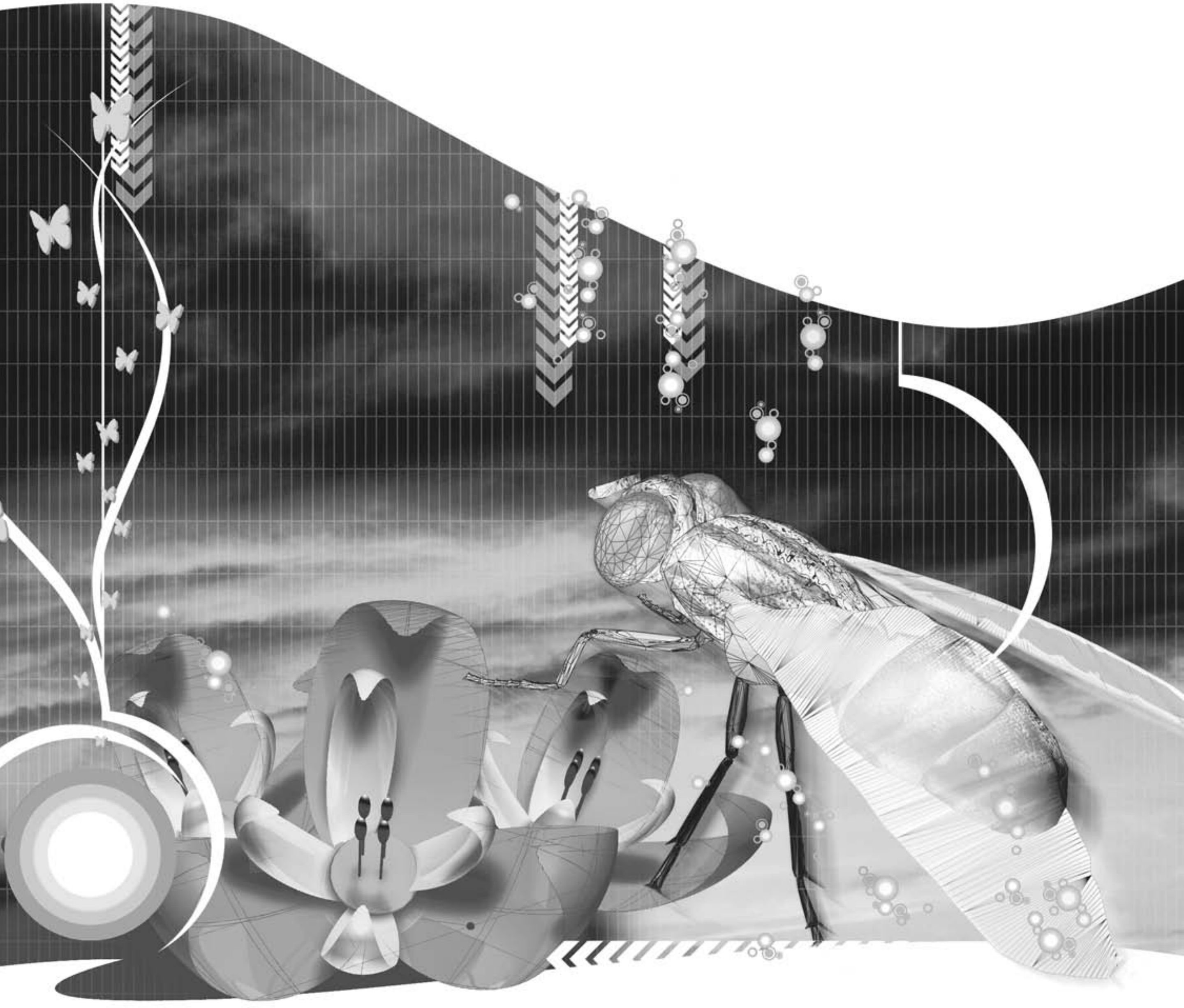
By breaking each task in your movie into several well-defined subtasks, you can keep your blocks of script relatively short and focused on specific goals. This makes it much easier to build your movie script in nice, easy-to-digest chunks. What's more, by nesting functions inside other functions, you can put as many levels of functions as you need between the overall goal of the movie and the individual actions that actually get the job done.

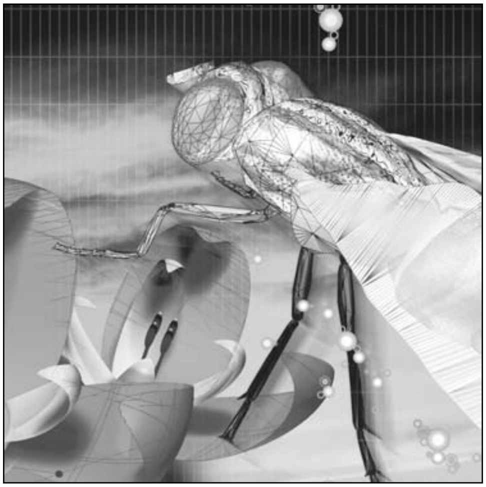
Let's recap the benefits of using functions:

- **Conciseness**: You define the function once, and you can call it as many times as you like.
- **Maintainability**: You change the function definition once, and you update its functionality throughout the movie.
- **Versatility**: You use arguments to apply the same function to different things.

Now that you've started to look at how you can structure the actions behind your scripts, you're ready to go a step further and bring data back into the frame. In the next chapter, you'll see how you can bundle functions *and* variables, helping you make scripts that are even more self-contained and reusable.

The practical benefits of all this stuff may not seem particularly obvious at this stage, since even the largest scripts you've written so far have been pretty small (by professional standards, anyway). All the same, it's obvious that you'll want to move on to larger projects, so keep at it and you'll soon be in a position to reap the rewards! Buckle up, now—Base Camp 1 is well behind you, and the slopes are starting to kick in. Take a break and prepare to unlock the ancient secrets of Mount ActionScript.





Chapter 7

OBJECTS AND CLASSES

What we'll cover in this chapter:

- Using objects to describe things with state and behavior
- Storing related collections of variables and functions
- Creating custom object classes
- Working with ActionScript's core built-in objects

We spent much of the last chapter discussing how functions can help give your scripts a nice, clear structure—especially when you choose them wisely and name them sensibly! One of the techniques you looked at dealt with nesting functions. That way, you can break down any complex task into a few slightly simpler subtasks, break down each of those into even simpler sub-subtasks, and so on, until each one reaches a level at which it needs just a few simple actions to do its job. You end up with lots of short, tightly focused functions, each playing its own special part in the overall movie script.

On their own, functions can be fairly useful, but they also have one or two drawbacks:

- A function can't remember anything for itself. Any variables you use set up inside the declaration will be created when the function starts to run and destroyed as soon as the function is done. If you call a function three times in a row, it will do exactly the same thing every time, because the third call doesn't know anything about the first two.
- If a function does make a lasting change to the movie, it's because it contains script that acts on things that are defined somewhere else in the movie. For example, the `swap()` function you considered for the hangman game refers specifically to a named button instance *and* a named movie clip. Neither of these things is part of the function, but the function itself is no use to anyone unless both the button and movie clip exist in the movie.

OK, but so what? You've used functions, and they worked fine. What's the problem?

The problem is this: as your movies get bigger and bigger, it gets harder to keep track of what's where. This makes it much more difficult to get away with writing functions like `swap()`, because they rely on things that may or may not exist somewhere out there in the murky depths of your Flash magnum opus. To help avoid this sort of thing, you have to make your code more *general*. You need to learn how to work with **objects** and **classes**.

Introducing objects and classes

Suppose you were having one of those deep, meaningful conversations concerning what reality is all about. Just what is stuff all about? What are the material objects all around us, and what is their relationship to each other? Well, this probably isn't the answer, but it's a good preamble and link into what we want to talk about, so here goes:

Objects are individual things such as

- My car, which has a unique license plate to differentiate it from others on the road
- The keyboard that I'm writing this book with, serial number KY24000BXXXA
- The weird latex frog toy that a friend gave me, which sits on top of my monitor and I've named Flash the frog

The individual objects around us are unique things that conform to a set of broader groups that defines them, for example:

- My car is a particular model and make, and this defines its basic features and function.
- My keyboard is an Apple Pro keyboard, and it looks and works like all other Apple Pro keyboards.

- There is a factory somewhere that makes loads of latex frogs according to a basic design. This design specifies all of them as being equally weird as Flash the frog, and they'll have the same trickery involved in them that makes Flash feel permanently wet when you touch him, but your hand is always dry.

These groups that objects are defined by can be called a number of things depending on what you're describing—species, model, or blueprint. However, they all broadly mean the same thing: **class**, which you can define as "a template that defines all objects within it."

You don't just have lots of different classes, though—you have **hierarchies** of them. For example, my keyboard is an Apple Pro, but it's also part of a larger class of electronic things called **input devices**, and **input devices** are part of another class you could define called **computer peripherals**.

So now you might be thinking, "Hey, you should be on talk shows that specialize in personal-relationship dilemmas for spouting the obvious like that! C'mon, what does any of this have to do with coding?" Well, quite a lot. The concept of individual objects and a hierarchy of classes that defines them underpins a set of modern coding techniques that many computer languages, including ActionScript 2.0, conform to, and these techniques go by the grand name of **object-oriented programming** (**OOP**).

There are several flavors of OOP. The OOP flavor Flash MX (version 6) used was a stripped-down one, with only predefined classes. It's referred to as ***prototype based****, and you won't consider it in this book. The OOP flavor Flash 8 uses concentrates on building classes and their interrelationships to define objects, and it's sometimes called* ***class based****.*

In fact, it's precisely *because* objects and classes of objects are familiar things from the real world that they're useful to us. Say you have a car. It could *be* a red Ferrari with a 2400cc engine. Lovely. Common sense tells you there are certain things it can *do*: start, stop, change gears, reverse, and so on. It's nice to imagine actually doing those things, but there's no great mental strain involved.

On the other hand, say you have a brick. It could *be* 2 pounds in weight, brown, made of clay, and part of the top of a wall. But it can't *do* much; it just sort of . . . sits there.

These two examples highlight the important characteristics of real objects: they're things that have **state** (all the things they *are*—where they are, what they look like, what they're made of, and so on) and **behavior** (all the things they can possibly *do* to change that state in some way). Because they're examples of objects that you've encountered in the real world, you already have a good idea of what these things can and can't be. Common sense tells you that you can't change gears on a brick (unless, perhaps, it's one of those lovely new LEGO blocks), so you're not likely to try!

Based on this idea, ActionScript objects give you another sort of descriptive container (like variables are for data and functions are for actions) that can *be* and *do* lots of different things, just like the objects you see around you in the real world. The *being* is taken care of by **properties** (for values that can vary, such as position), while the *doing* is defined in terms of functions, which are now called **methods** in the OOP system.

Once you start digging around behind the scenes, you'll find that objects and classes are fundamental to everything that goes on in Flash and ActionScript. You don't need to create them all from scratch, as you've done in the past with variables and functions, because Flash already has plenty of built-in classes for you to work with. You can, however, create new classes if you need (for example) to model weird latex frogs via ActionScript. Commonly in web design you don't need that many frogs, but you do need a class of objects that you can click and create web navigations with. Such a class might be called `Button`. You might also want a class that represents timelines full of tweens and graphics, and call it `MovieClip`. Sounding familiar yet?

Objects are very much fundamental to programming, and people call them different things. Another common name is **instance**. Ah! Now we're back on track!

There are two final things you need to fit into your linkage between Flash OOP, the things we've discussed so far in the book, and the real world, and you'll have cracked it: **type** and **abstraction**.

Type and object-oriented programming

Typing is simply the process of differentiating between classes and not allowing different classes to mix. For example, if you were running a zoo, you would try to treat the birds differently from the fish. You wouldn't get far if you put all the fish in a big cage with perches and plenty of unobstructed space to fly around in, and fed them seed plus a little fat in winter.

In the same way, objects need to be treated differently depending on what they are, and this is what type is all about. You use type to tell Flash what each object is, so that if you try to keep your hummingbirds in a 6-foot-deep tank containing saltwater, gravel, and a faux underwater cave, it can tell you there's a mismatch between the type (bird) and the context you're using it in (glug glug).

In Flash, type doesn't give you faster or neater code, and you don't have to use it if you don't want to. Although in the real world you know birds and underwater grottos don't mix, in the world of Flash objects, what does and doesn't go together can get a little technical and may have no real-world analogy you can use to sort it out in your own mind. Typing your code allows Flash to keep track of your data zoo and keeps all your information in its own cage, fed by the appropriate functions and watered by the correct interfaces.

Strict typing is all about maintaining the integrity of your data so that the different sorts of information (or, more specifically, the different classes of information) are kept separate.

Classes, generalization, and abstraction

So what use is this class thing? This object-class philosophy-of-stuff seems very neat and sensible, but what is it good for? What if you instead defined a code model based on the sayings of Forrest Gump's mom? Some of that was sensible as well. Well, the thing about thinking in terms of classes is that you can generalize and abstract. Let me explain.

There's a cool thing about my Apple Pro keyboard, and that is that my computer *isn't a Mac*. It's a PC. Because I know from experience that an Apple Pro keyboard is the best keyboard for the sort of constant typing that a full-time author has to do, I use it in place of the decidedly variable keyboards PCs come with. The reason that the Apple Pro keyboard works on my PC is that although it has basic classes (model, Apple-specific keyboard layouts and keys, and so on) that are different from PC keyboard models, if you go back far enough, PC keyboards and Mac keyboards have the same master class: they're all in the same **computer input devices** class. Among other things, input devices in this master class have the same connection: **universal serial bus** (**USB**). So I can plug my Apple Pro keyboard into my PC, and the PC won't say, "Uh oh, that's a Mac object—I can't use it. Sorry, I think I feel a blue screen coming on." Instead, it will say, "Oh, OK, that's a standard USB keyboard object. Let me get the general interfaces to identify and run that from the Windows CD-ROM, and you can get going on your next opus almost immediately."

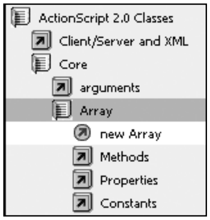
I guess a programming philosophy based on Forrest Gump sayings would be cool because the rules would be funny in a wry, "idiot talks rubbish but there's a deeper postmodern meaning in what he says" kind of way, but they wouldn't give us Apple Pro keyboards that work anywhere or general ActionScript that can be used for many different purposes, because the underlying code philosophy is about generalization.

Seeing arrays in a new light

Back in Chapter 3 you started playing with arrays. As you've seen since then, arrays can be terribly useful for storing lots of variables in one place, and with the power of loops added in, they're clearly a force to be reckoned with.

Well, you ain't seen nothing yet!

The humble array happens to be one marvelous example of an ActionScript class. You already know it as a container that holds lots of variables (rather like a real-world numbered list). It also has a few tricks up its sleeve that you will find invaluable throughout your career as a Flash 'scripter. You can see what's on offer by taking a look in the Actions toolbox (on the left side of the Actions panel), where you'll find Array listed under ActionScript 2.0 Classes ➤ Core, with four entries beneath it in the hierarchy.



In the simplest possible terms, you can think of an object as a bunch of related values, constants, and functions, wrapped together to make a neat, self-contained building block that you can use in your movie script. When you're dealing with classes, there are special technical names you give to the three parts of an object: the values are called properties, the constants retain the same name, and the functions are referred to as methods.

The Array constructor method

The first of the `Array` methods is `new Array`, and this is a special sort of action known as a **constructor**. Why? Because it's what you use to *construct* an `Array` instance (or, to use the alternative term, an object) in the first place. You've seen this used quite a few times already, for example:

```
letters = new Array("f","l","a","s","h");
```

This line tells Flash to construct a new object of type `Array`, give it five string elements (with values as shown), and store it in the container `letters`. Of course, that's pretty much as you understood it before—or at least, I hope it is!

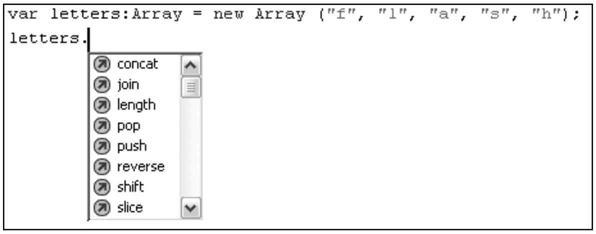
If you want to create a typed variable to avoid the “kestrel found dead in tropical fish aquarium,” zoo-shocker type errors, you would use something like this to do the same thing:

```
var letters:Array = new Array("f", "l", "a", "s", "h");
```

> *As well as preventing you from mixing data, using strict typing has another advantage: Flash knows you’re talking about an array without having to run the code to find out. This means that it will use code hinting to show you all possible properties, methods, and constants as soon as you type letters. It will do this without your needing to use the* `_array` *suffix that nontyped versions of ActionScript (those before ActionScript 2.0) require.*
>
> *Also worth noting is that when programmers use the term “an array,” they mean an instance that is an array. When programmers say “Array,” they mean the class that defines what an array is. When referring to the* `Array` *class, you always use an uppercase letter “A” in the term, both in the code and in written text.*

If you use the typed version of your constructor, you’ll see that hints will work straight out. If you use the seemingly quicker nontyped version, you’ll have to change the array name to `letters_array` before Flash knows you’re talking about the `Array` class, so you’ll probably get more mileage on the “save typing” front by using strict typing.

For example, if you type the code shown in the following image, Flash will know `letters` is an array by the fact that you’ve defined its type as `Array`. This means that as soon as you key the text letters. (`letters` followed by a dot) into the Script pane, Flash will know the next thing you need to add is a method or property of the `Array` class, and it will show you a drop-down menu with the possible choices.



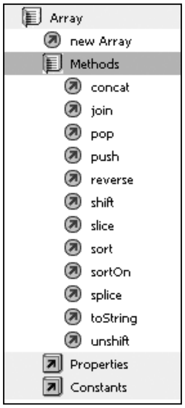
```
var letters:Array = new Array ("f", "l", "a", "s", "h");
letters.
```

If you don’t use strict typing (i.e., you don’t include the first line of the preceding code), the only way you can tell Flash that `letters` is an array is by calling it `letters_array`.

Other Array methods

Now take a look in the Methods book, and you'll see a list of all those tricks mentioned earlier:

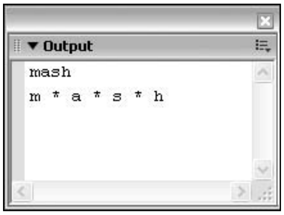
The range of methods here perform actions from removing an array element to sorting an array. Let's take a look at some of them.



Array
new Array
Methods
concat
join
pop
push
reverse
shift
slice
sort
sortOn
splice
toString
unshift
Properties
Constants

The join() method allows you to join together all the array elements to form a string. It takes one argument, which is a specified string delimiter to place between each element and the next. Here's an example followed by its output:

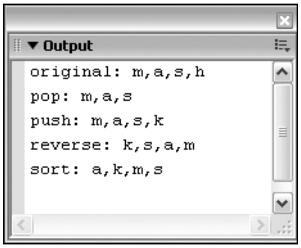
```
var my_str:String = "";
var my_array:Array = new Array("m", "a", "s", "h");
my_str = my_array.join("");
trace(my_str);
//
my_str = my_array.join(" * ");
trace(my_str);
```



```
mash
m * a * s * h
```

Just like the functions you saw toward the end of the previous chapter, the join() method simply returns a value—a string, in this case—and has no effect on the array on which it operates. That's not always the case though, as many array methods, such as pop(), reverse(), and sort(), actually affect the structure of the array. Change the code as shown, and the array is affected profoundly:

```
var my_array:Array = new Array("m", "a", "s", "h");
trace ("original: " + my_array);
//
my_array.pop ();
trace ("pop: " + my_array);
//
my_array.push ("k");
trace ("push: " + my_array);
//
my_array.reverse ();
trace ("reverse: " + my_array);
//
my_array.sort();
trace ("sort: " + my_array);
```



```
original: m,a,s,h
pop: m,a,s
push: m,a,s,k
reverse: k,s,a,m
sort: a,k,m,s
```

When passing an array to the trace() *function, the whole array is outputted as a comma-separated list in the* Output *panel, as shown in the preceding screenshot.*

As you can see, the four methods used here dismantle the array beyond all recognition! Here's what each of these does in a little more detail:

- pop(): The pop() method simply removes the last element from the array. So when it was applied to our original array structure, it left us with the first three elements: m, a, s. Adversely, the shift() method will remove the first element from the array.
- push(): The push() method allows one or more elements to be added to the end of the array. To add more than one element to an array, separate the new elements with commas like so: push("k", "e", "d");.
- reverse(): This method reverses the elements of the array.
- sort(): This method is used to sort an array by numeric or alphabetical order. By default (i.e., without arguments), the array is ordered in ascending order. Be aware that this method has a number of idiosyncrasies. One of these is that the uppercase alphabet precedes the lowercase, so the array a, B, c, D would be sorted in this way: B, D, a, c. However, arguments passed to the sort() method allow control over how the elements are sorted.

Given all this information, how about a quick challenge? Let's start with array elements reading m, a, s, h and we want it to end up as f, l, a, s, h. How would you do it? Take a second to think about it before we tell you the answer . . .

Although we've highlighted only a small number of the Array methods here, there isn't time to go through all of them or we'll never get on to the cool stuff! If you're curious, though, the ActionScript 2.0 Language Reference (accessible via Flash's Help panel) does a pretty good job of describing what the rest of the methods do if you feel the need to find out more. The important thing is that you know how to use the methods: simply write the name of the array you want to work on, add a dot, and write the name of the method you want to apply, followed by a pair of parentheses. Sometimes you'll want to add an argument or two between the parentheses, but more often than not, you'll just leave them empty for the simpler methods you'll be using as a beginner.

Oh, here's one way of completing the challenge, by the way:

```
var my_array:Array = new Array("m", "a", "s", "h");
//
my_array.shift ();
my_array.unshift ("f", "l");
```

OK, OK, we cheated a little by using the unshift() *method (which we accidentally didn't mention earlier). The* unshift() *method simply allows elements to be added to the start of an array.*

Array properties

Look in the Properties book, and you'll find a single entry called length. If you remember back to the hangman game in Chapter 5, this is just the property we used to find out how many array elements to loop through:

```
var elementsInLettersArray:Number = letters.length;
```

No parentheses here, because it's a property, not a method. OK, been there, done that. But is that *it* for the array properties? What about all the elements? Surely there are array properties too? Yes, there are. But think about it for a moment, and you'll see why they aren't among the built-in properties listed in the Actions toolbox. It's because they're *not* built in! All the elements you add to an array are **custom properties**. In other words, they don't need to exist until your script specifically defines them, so they aren't part of ActionScript's general definition of what an array is.

Eagle-eyed readers will notice that they've used exactly the same dot notation to run actions, functions, and now methods. Very spooky!

Creating classes and objects (instances) in Flash

As discussed so far, a class is really just a template for a particular type of object (whereas "type" actually means the same thing as when you "strictly type" a variable) that defines how the object should be constructed, what the object should be able to do, and so on. In the same way as a physical building always starts off as a blueprint on an architect's drawing board (or on an AutoCAD-enabled desktop), an object always begins life as a class.

The class contains a list of the common characteristics that all objects constructed from it should have. For example, imagine you have a class called Plant. It might have a template that includes the following:

- Name of the plant
- Height of the plant
- Color of the plant's flower
- Flowering month
- Recommended soil type

This might not seem particularly useful in a Flash movie, unless, that is, you suddenly landed a big contract with an international chain of garden centers, found yourself commissioned to design an interface for the "create a garden" page on the company's website, and needed a way to keep track of all the different plants the company sells. You never know when this sort of thing might come in handy!

Take a quick browse through the Actions toolbox, and you'll notice that all the built-in ActionScript class names (Array, Boolean, Date, Function, and so on) begin with an uppercase letter. This helpful convention reminds you whether you're dealing with a class or an instance in your script.

Instances

Just as a single set of blueprints can be used as the basis for lots of physical buildings, one class can be used to produce lots of distinct instances of Array class objects. Any object instance based on the Plant class would feature a corresponding list of values (or properties). So, you might have an object instance called myFirstFlower (of class Plant) that includes the following properties:

- daffodil
- 20cm
- yellow
- March
- normal

Now you could use this information to cover a virtual garden with daffodils and show them flowering ready for St. David's Day (a day celebrating the patron saint of Wales) in March (or dying if the soil turned out to be too acidic). Another instance might store information about sunflowers, another one might contain information about lilies, and so on.

The Object object

All the classes you've looked at so far have been fairly specific, such as Array and MovieClip. A more general class is the slightly confusingly named Object class.

Wha . . .? A class called Object? What's an object of this class called? Um . . . an Object instance or, if you really like it silly, it's called an Object object (note the "O" and "o").

Right. Well, there is method to this madness, as you'll see. Suppose you want to create a single object that defines a plant. To define an object to hold some information (properties) about daffodils, you would use something like this:

```
var myFirstFlower:Object = new Object();
myFirstFlower.name = "daffodil";
myFirstFlower.height = 20;
myFirstFlower.flowerColor = "yellow";
myFirstFlower.flowerMonth = "March";
myFirstFlower.soilType = "normal";
```

Here you create another object from the built-in Object class, just like you did with the Array class before. You then step through each of the properties you want and tell Flash what values to put in which slots, just like when you define variables, except all the value containers are properties inside the myFirstFlower object.

Viewing an object in Flash

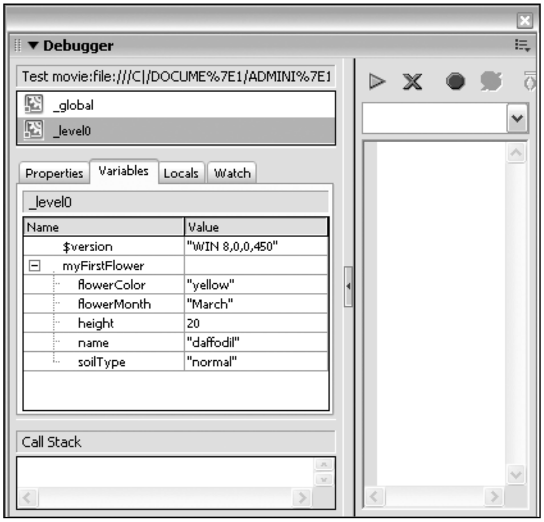
You can view the structure of the object `myFirstFlower` if you *debug* your movie.

1. Assuming you've entered the code in the preceding section (attach it to the first frame of a new FLA), select Control ➤ Debug Movie to enter test mode with the Debugger panel active.

> *As your code starts handling more and more complex data, you'll begin to realize that the content the user sees on the stage is less important to you as a programmer than the unseen stuff that's going on. It's the data created and manipulated by your ActionScript that's the major thing in your SWF. The* Debugger *panel is a window into this data and an important tool for the ActionScript developer.*

2. The movie will appear paused. Click the green "Play" button (it's actually labeled Continue but looks like Play to us) on the right side of the window to start the movie. You'll see nothing on the stage when you do this. Your code defines `myFirstFlower` but does nothing with it or the stage.
3. You can look at the definition of `myFirstFlower` via the Debugger panel. You first need to get the left side of the Debugger panel to show the three panes shown in the following image. If you've never used the Debugger panel before, the middle (tabbed) pane will be hidden. To reveal it, click-drag the top of the Call Stack (lower panel) downward.
4. To find your object, you need to specify where it is (i.e., the *scope* of `myFirstFlower`). In the top panel, click _level0. This is another name for the root timeline and is the scope of your object.

   The Variables tab name is a little misleading, because this view also shows objects. Your object is the second one down (the first one, $version, is a number containing the version of the Flash player currently being used to display your SWF).
5. Click the plus sign (+) next to myFirstFlower to see the hierarchy of properties you've created in your script. The Value column tells you the value of each property.



Test movie:file:///C|/DOCUME%7E1/ADMINI%7E1

_level0

Name	Value
$version	"WIN 8,0,0,450"
⊟ myFirstFlower	
flowerColor	"yellow"
flowerMonth	"March"
height	20
name	"daffodil"
soilType	"normal"

As you can see from the preceding image, the object is a tree diagram, with the properties branching from the object itself. The usefulness of an object is tied into this tree: it defines a ***structure*** *for your object, and this structure makes it easy for code to read the many property values. You may hear objects called* ***structured data****, and it's this tree that the term "structured" refers to.*

You could have defined the data associated with `myFirstFlower` *as a set of separate variables, but that would result in* ***unstructured data****, something that is easy to do if you want only one flower, but if you want 50,000 different flowers, you'll find it difficult to manage your code.*

You'll see the advantages of structured data when you start to use methods shortly.

Constructors

`Object` is a totally stripped-down class that has the bare minimum of properties and methods (none of which you really need to know about at this level) to let you add in properties like those in the preceding section. It therefore gives you a clean slate on which to build your own objects.

The only trouble with the code as it stands is that you've only made an instance of the `Object` class. There's no way to create a second instance (called `mySecondFlower`, perhaps) with the same set of properties without going through all of these lines setting up properties—and doing it again, and again, and again, for every new flower you decide to add in.

Instead, you need to define a `Plant` constructor function, like this:

```
function Plant (name, height, color, month, soil) {
  this.name = name;
  this.height = height;
  this.flowerColor = color;
  this.flowerMonth = month;
  this.soilType = soil;
}
```

You're probably wondering what all that business with `this` is all about. `this` is a keyword that refers to whichever object happens to have called your function. At this point it has no meaning, but as soon as you try plan A again . . .

```
var myFirstFlower:Object = ➥
  new Plant("daffodil", "10cm", yellow", "March", "normal");
```

This time, Flash uses the `Plant` function to construct an object called `myFirstFlower`, with the properties `name`, `height`, and so on. The arguments you use make sure they're all given values to hold: `"daffodil"`, `"10cm"`, and so on. The advantage of doing it this way is that if you want to create another object instance of `Plant` (say, `mySecondFlower`, a 15 cm red tulip that grows in normal soil and flowers in June), you'd just need one more line to define it.

```
var mySecondFlower:Object = ➥
  new Plant("tulip", "15cm", "red", "June", "normal");
```

> *In fact,* `Object` *is an extremely important class for ActionScript, as it's the class that all others are based on. It defines the basic rules for all ActionScript objects, and one of these is that it lets you add in properties as you go along. The great thing about working with* `Object` *is that it won't bog down your* `Plant` *objects with features that they don't need. It would be foolish to use the* `Array` *class as the basis for an object like this. There would be no point in having* `sort()` *and* `reverse()` *methods for an object whose sole purpose is to describe a plant—what on earth would those methods do? No, the* `Object` *class gives you a lean, mean instance that stores several pieces of associated data, ready for you to use further down the line.*

You can now read and write property values in `myFirstFlower` just as if it were any other object:

```
// traces "daffodil"
trace(myFirstFlower.name);
// traces "10cm"
trace(myFirstFlower.height);
// changes value of height property to "15cm"
myFirstFlower.height = "15cm";
// now traces "15cm"
trace(myFirstFlower.height);
```

Usually, though, you don't want to be messing about with properties—that would just make them like overly complicated variables. This is where methods come in. They do all the work of manipulating the properties for you. Let's see how that would work. Change the listing as shown:

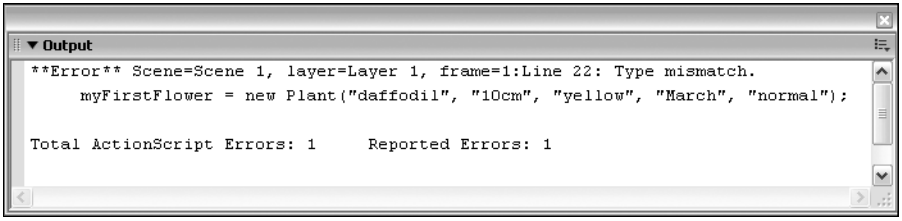
```
function Plant(name:String, height:Number, color:String,➥
 month:String, soil:String):Void {
  this.name = name;
  this.height = height;
  this.flowerColor = color;
  this.flowerMonth = month;
  this.soilType = soil;
  this.listPlant = function() {
    trace("name: "+this.name);
    trace("height: "+this.height);
    trace("color: "+this.flowerColor);
    trace("month: "+this.flowerMonth);
    trace("soil type: "+this.soilType);
  };
  this.plantInSummer = function() {
    if ((this.flowerMonth.toLowerCase() == "june") || ➥
       (this.flowerMonth.toLowerCase() == "july") || ➥
       (this.flowerMonth.toLowerCase() == "august")) {
      return true;
    } else {
      return false;
    }
  }
}
```

This time, you also type your constructor to make your code more able to detect errors, and you add two functions, `listPlant()` and `plantInSummer()`, inside the constructor.

The best way to see how this code works is to try it out. Add the following definition and then run the movie:

```
var myFirstFlower:Object = ➥
  new Plant("daffodil", "10cm", "yellow", "March", "normal");
```

Whoops—you've done something wrong!



```
**Error** Scene=Scene 1, layer=Layer 1, frame=1:Line 22: Type mismatch.
     myFirstFlower = new Plant("daffodil", "10cm", "yellow", "March", "normal");

Total ActionScript Errors: 1     Reported Errors: 1
```

The plant height of `firstFlower` is given as `"10cm"`, which is a string. The constructor expects a number at the head of the constructor:

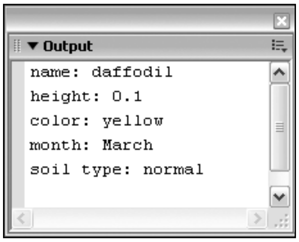
```
function Plant(name:String, height:Number, color:String, ➥
   month:String, soil:String):Void {
```

Change the line as follows:

```
var myFirstFlower:Object = ➥
  new Plant("daffodil", 0.1, "yellow",  "March", "normal");
```

Better—ah . . . except that it does nothing! Well, you've actually done something: you've created an object that defines `firstFlower`. What you aren't doing is using it. Now add the following line:

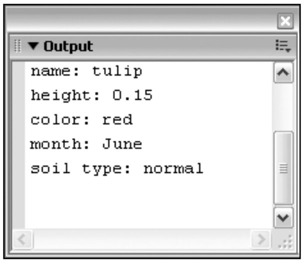
```
var myFirstFlower:Object = ➥
  new Plant("daffodil", 0.1, "yellow",  "March", "normal");
myFirstFlower.listPlant();
```



```
name: daffodil
height: 0.1
color: yellow
month: March
soil type: normal
```

The method listPlant() lists all the properties of your object. Add the following lines to try it with another object, mySecondFlower:

```
var mySecondFlower:Object = ➥
  new Plant("tulip", 0.15, "red", "June", "normal");
mySecondFlower.listPlant();
```



```
name: tulip
height: 0.15
color: red
month: June
soil type: normal
```

Finally, try the other method, plantInSummer(). This will look at the object properties to see whether it can be planted in the early-to-mid summer months. It will return true if it can and false if it can't. Add the following lines at the end of the current script:

```
trace(myFirstFlower.plantInSummer());
trace(mySecondFlower.plantInSummer());
```

You'll see false for the daffodil and true for the tulip.

So what can objects do that variables can't?

- They allow you to create data with custom structures. Your plant data is a far more complex set of data than simple, single-value numbers or strings, and it's a little more advanced than Array lists. In real-world solutions, data is always complex in the same way, and objects are one very good way of managing this.
- The plant objects can handle their own information via methods. Rather than write custom code to look at whether you can plant in summer, you simply ask the object itself. The object uses its methods to act on its properties, and it gives you the answer you want via stuff (properties and methods) that's part of the object itself. In short, the data and the means to handle it are both encapsulated in one "thing": the object.

A word of warning: The form of object you've just looked at is an example of an object that uses the predefined Object class. You don't create a class here; rather, you create a constructor function. The problem with it is that each object gains its *own* template rather than all your objects having the same template (which is what happens with class and true OOP).

Later on in the book, you'll look at the Flash 8 class-based way of handling complex data, but for now, you need to take away the fact that objects consist of methods and properties, arranged in such a way that they become extremely useful for representing (and manipulating) data.

Objects, objects, everywhere . . .

So far, you've seen how the new constructor coupled with the class name Array will make an instance of the Array class. Likewise, you can pair new with the class name Object to create an instance of the Object class, and you can even write constructor functions that new can use as templates (but not quite full Flash 8 classes), from which to create custom objects. So is this new business the only way to create object instances?

Actually, no it isn't. Most of the time, ActionScript does lots of work behind the scenes so that you can use objects without even realizing it. For example, it's pretty easy to create a variable:

```
var myName:String = "Karl";
```

You probably never realized it before (but you've probably picked up a few hints!), but variables themselves are based on classes. When you create a string variable, it inherits various properties and methods from a class called String, which you can find listed under ActionScript 2.0 Classes ➤ Core ➤ String.

As you might be coming to expect, the String class has a constructor method new String and a length property just like the one you saw for the Array class. It also has a bunch of methods that can be (and have been) used to manipulate the characters in the string. You may remember using the toLowerCase() method as part of your password-protected movie back in Chapter 4. Well, this is where it started life—as part of the String class.

All the methods of the String class (as found in ActionScript 2.0 Classes ➤ Core ➤ String ➤ Methods) are things that you might want to apply to the letters in a string. Many of them are similar (if not the same) as the Array methods you saw earlier, letting you mess around with letters in a string in the same way as you messed around with elements in your array.

Lurking objects

There are certain ActionScript classes that, for one reason or another, you don't need to construct for yourself before you put them to use.

> *The technical term for these classes is **static**. A static class doesn't need to be defined, usually because there's only one. For example, there's only one stage in Flash, and your computer will have one mouse pointer on the screen and one active keyboard. In each case, you have no reason to create more than one object to work with any of these. To keep things simple, Flash defines the one instance that you'll need.*

One of these classes is called Math, and it's designed to work a bit like a simple calculator. It has methods that let you do complex mathematical operations (pow, sqr, sin, and cos) and generate random numbers, and properties that store "useful" mathematical constants (PI, anyone?). No, don't laugh! Rest assured, when you start wanting to put 3D effects into your movies, you *will* find them invaluable.

Unlike most calculators, though, Math doesn't have a memory, so in many ways, it acts like a library of math commands that you can use whenever and wherever you need them. Since all the properties are constants (i.e., they all have values that can't be altered), it doesn't make much sense to construct instances of Math. They'd all be exactly the same, and you wouldn't be able to change any of their properties, so what on earth would you do with more than one instance of the Math class? The answer is, you don't need more than one. You access the single Math class directly like this:

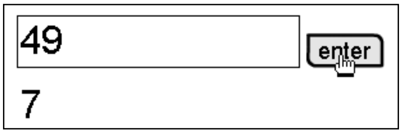
```
output1 = Math.sqrt(input1);
```

Notice that you don't have to define new Math() before you start using it.

You can wire this up by using the file inputOutput.fla you created back in Chapter 3. Open this FLA and change the script to the following:

```
enter_btn.onRelease = function() {
  output_str = Math.sqrt(input_str);
};
```

This takes input_str as an argument in the sqrt() method, which works out the square root of its value (i.e., it figures out the number you have to square to get input_str).



Text fields always assume a string value, unless you tell them otherwise, so (although in this case your code works because the Math class is clever enough to figure it out) you should really use this:

```
enter_btn.onRelease = function() {
  output_str = Math.sqrt(Number(input_str));
};
```

Number() makes sure the input value is taken to be a number. Of course, that doesn't stop you from seeing NaN (not a number) as an answer if you click the enter button without entering a number, or if you enter a string such as "cat:", but you already know that from Chapter 3, and the fact that you can limit the inputs to what you want (such as numerals only) . . . see Chapter 3 for a refresher.

Another object you access directly via its class name is Stage. As mentioned earlier, the reason this is a static class is because there's only one stage, so you go ahead and start using Stage as if it was always there, because, um . . . it was.

So what use is this Stage object? Well, say you want to know the current dimensions of the stage. You can call up the width and height properties like this:

```
xDimension = Stage.width;
yDimension = Stage.height;
```

Big deal! You knew that from the Property inspector. But say, for example, you want to make a movie that shows a ball bouncing around on the stage, within the *confines* of the stage. Sure, you could set up a couple of variables (xLimit=550 and yLimit=400) telling it where the far edges of the stage were.

So what then if you *changed* the size of the stage, but *forgot* to update the variables? Bingo—busted movie! It's much safer if you use Stage.width and Stage.height to set your limits, because then they're both updated for you automatically.

You can find the Stage class listed in the Actions toolbox under ActionScript 2.0 Classes ➤ Movie. Listed with the Stage class, you'll see one or two familiar names. Yes, these objects really are *everywhere*, including the stage itself!

Now that you've learned how to work with purely ActionScript-based objects, you're all set to discover what objects have to offer you in the graphical realm.

Making a show reel

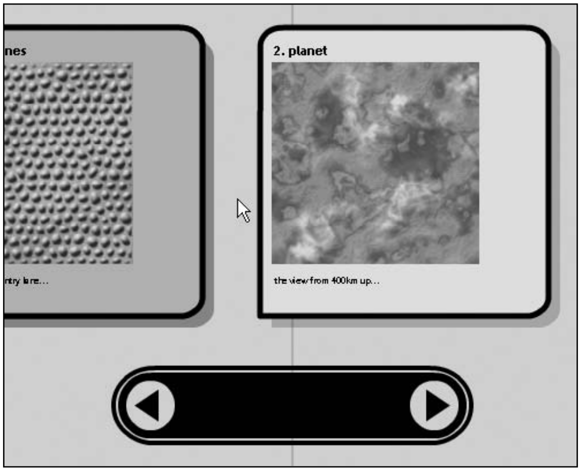
Lots of sites out there use dynamic animation in a *functional* way. That is to say, the animation serves a purpose rather than just acting as eye candy. Navigation systems are perhaps one of the most useful things you can do with ActionScript. In this example, you're going to build an image gallery using a sideways scrolling motion to navigate through a show reel. You could use it for anything you want, whether that's showing off your holiday pictures or presenting an online portfolio.

In the process, you're going to use objects as a means of passing information to functions. You can find the finished show reel, saved as showreel.fla, in the download for this chapter. Copy it to a directory on your machine and put copies of the three accompanying JPG files (cobblestones.jpg, planet.jpg, and water.jpg) in the same directory. Run the SWF to see what it does.

It looks like these objects aren't content with meddling about behind the scenes! They're making a bid for freedom and starting to take over the stage itself. Is nothing sacred?

Credit and thanks for these objects must go to Photoshop masters Scott Hamlin and Scott Balay (of Eyeland Studio, www.eyeland.com*) from whose book* 40 Photoshop Power Effects *(friends of ED, ISBN: 1590592026) came the spiffing techniques whose end results you see here.*

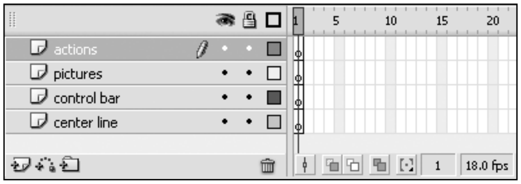
It's very simple to use: just move the pointer toward the left, and the show reel will scroll over to the left. Move the pointer toward the right, and the show reel will scroll back to the right. The scrolling increases in speed as you move the cursor farther from the center line.



Remember the movies you made back in Chapter 1, when you looked at a tiny script that could make various objects (including a car, a bat, and a Starfleet SpaceCruiser) follow the x-coordinate of the cursor? Well, this is essentially the same thing—just a little more elaborate. The original ball scroll was done with just two lines of ActionScript, and this effect doesn't take many more. This is the type of site navigation that just isn't possible with timelines; you have to start making use of ActionScript as a major part of your site.

Let's start building the show reel movie. First off, you need to prepare a few movie clips and give yourself something to work with on the stage.

1. Create a new Flash document and give it a total of four layers in the root timeline. You should call the layers actions, pictures, control bar, and center line. We've left the stage size as default (550 × 400 pixels), upped the frame rate of 18 fps, and given it a pale blue background (#CECFFF), so that it doesn't look quite so stark.

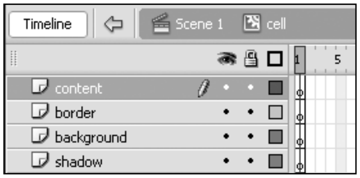


2. Next, you'll add the graphics that will help guide your users about the stage. We've used a vertical line to mark the middle of the stage (275 pixels across, on layer center line) and a chunky black bar to show in which direction the show reel will scroll (on layer control bar).

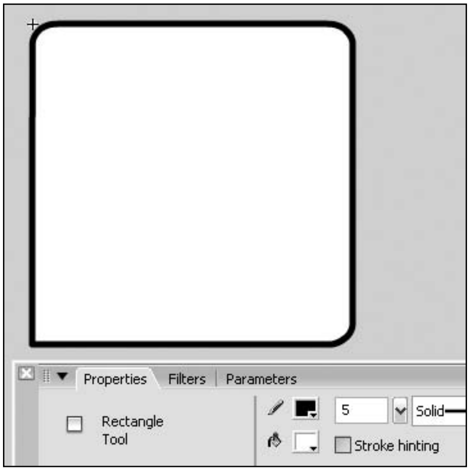
> *Note that the images themselves aren't built into the movie. As long as you have the files* `cobblestones.jpg`*,* `planet.jpg`*, and* `water.jpg` *in the same directory as the movie, Flash can pull them in when it needs them. It doesn't even matter what the images are—as long as you use the same filenames, you can replace the chapter's images with your own pictures and the movie won't miss a beat.*

Neither graphic actually *does* anything, so you won't have any more to do with them and you can lock their layers if you wish. Now for the important part: You're going to start building the show reel. It will consist of a long string of individual cells, with each one holding a single picture and information about the picture.

3. Use Insert ➤ New Symbol to define a new movie clip called `cell`, and give it four layers called content, border, background, and shadow.



4. Select the background layer and add a white square with a 5 pt black border using the Rectangle tool. You can use the Set Corner Radius button to give it curved corners if you like that sort of thing. It should be 250 pixels wide, 250 pixels high, and positioned with its top-left corner at X:0, Y:0.



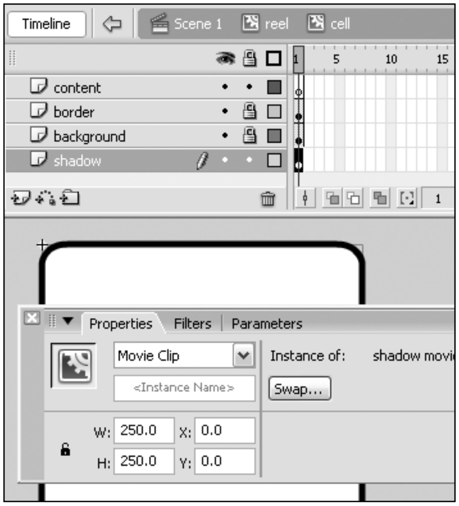
5
Solid

> *You can create a square by holding down the Shift key when you drag out with the* Rectangle *tool.*

5. Select the border, and use Edit ➤ Cut *and* Edit ➤ Paste in Place to move it into the layer called border.

6. Select the background layer again and copy the white fill you left there to the clipboard using Ctrl+C/Cmd+C. Paste it in place into the shadow layer using Ctrl+Shift+V/Cmd+Shift+V.

7. While the newly pasted shape is still selected, press F8 to convert it into a movie clip symbol, calling it shadow movie. Now lock the background and border layers so you don't accidentally change their content.

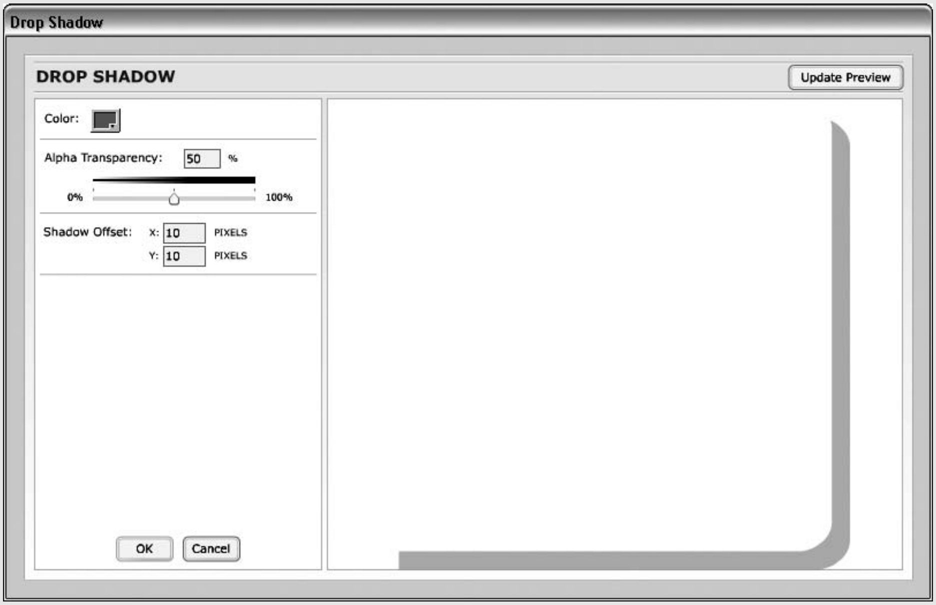


Movie Clip
<Instance Name>
Instance of: shadow movi
W: 250.0 X: 0.0
H: 250.0 Y: 0.0

You're now going to add a drop shadow filter to this movie clip instance, so that it gives the effect of the cell floating above the background.

8. Select the shadow movie instance on the stage (good thing you locked the layers so you can get it from underneath!) and select the Filters tab of the Property inspector.

Flash 8 Basic users might have already noticed the absence of the Filters *tab, so you'll be limited to using the Drop Shadow Timeline Effect. To do this with the* shadow movie *instance selected, select* Insert ➤ Timeline Effects ➤ Effects ➤ Drop Shadow. *In the dialog that appears, change the settings to reflect those in the following screenshot:*



Color:

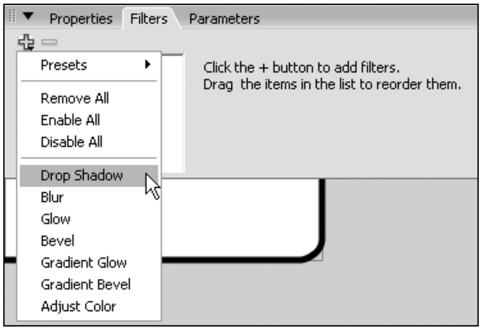
Alpha Transparency: 50 %

Shadow Offset: X: 10 PIXELS

Y: 10 PIXELS

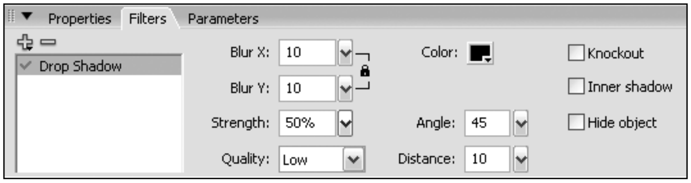
Click the Update Preview *button (top right) to view how the drop shadow effect will look. If you're happy, click the* OK *button to apply the Timeline Effect to the* shadow movie *instance. Feel free to now skip to step 11.*

9. Now click the + button to view the drop-down selection of filters. Select Drop Shadow from the list:



Click the + button to add filters.
Drag the items in the list to reorder them.

In the text fields that appear, specify the settings as shown here:



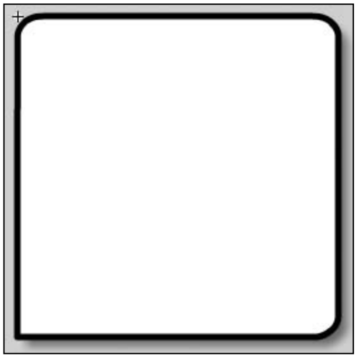
Properties | Filters | Parameters

Drop Shadow

Blur X:	10	Color:	■
Blur Y:	10		
Strength:	50%	Angle:	45
Quality:	Low	Distance:	10

□ Knockout
□ Inner shadow
□ Hide object

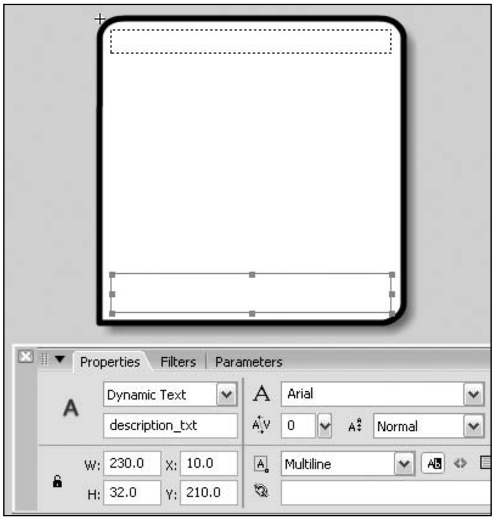
10. Now click away from the movie clip to view your drop shadow. Pretty groovy, huh?



11. Unlock the background layer and select it. Now press F8 to convert this layer's contents to a movie clip. You'll use this movie clip to give your show reel cells some color. Use the Property inspector to give it the instance name bgnd_mc.

12. Now select the content layer and add a pair of dynamic text fields, one at the top and one at the bottom.

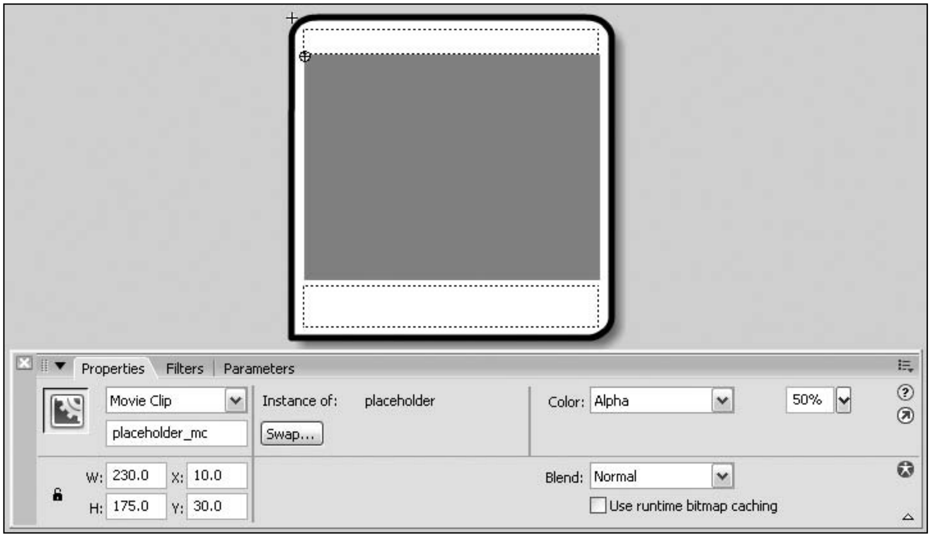
    Use the Property inspector to name the text fields name_txt and description_txt, and ensure that the font color for both is set to black. The lower one should also be set to Multiline, so that you can put in lots of nice descriptive text about your images.



Properties | Filters | Parameters

Dynamic Text
description_txt
Arial
0
Normal
W: 230.0 X: 10.0
H: 32.0 Y: 210.0
Multiline

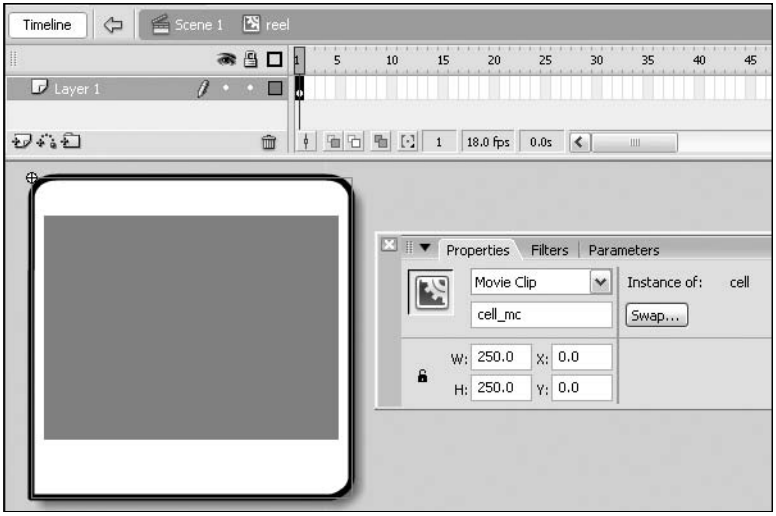
13. Still on the content layer, add a black box in the space between the two text fields. Press F8 to convert it to a movie clip called placeholder with a top left registration point. Then use the Property inspector to set the alpha to 50% using the Color drop-down. Set its instance name to `placeholder_mc`.



Movie Clip
placeholder_mc
Instance of: placeholder
Color: Alpha 50%
W: 230.0 X: 10.0
H: 175.0 Y: 30.0
Blend: Normal
□ Use runtime bitmap caching

Now that you've set up the individual cell as a movie clip, you'll do the same for the complete show reel.

14. This part is a lot simpler than the cell was. You just need to create a new movie clip called `reel` (Insert ➤ New Symbol), and inside your new, blank movie clip, put in a single instance of the `cell` movie clip. You should call the instance of the `cell` movie clip `cell_mc`.



1 18.0 fps 0.0s

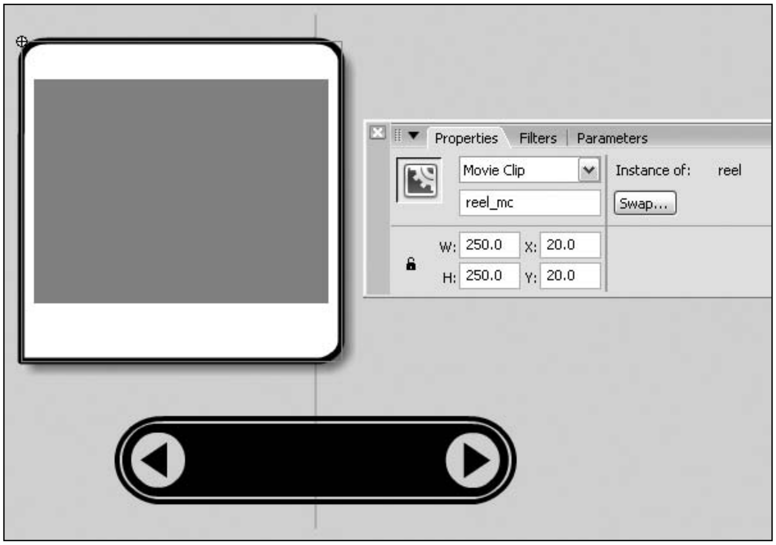
Movie Clip
cell_mc
Instance of: cell
W: 250.0 X: 0.0
H: 250.0 Y: 0.0

So now I imagine you're screwing up your face and saying, "What—just one instance? The FLA we looked at the start had *three* images in it . . ."

Yes, just the one instance. You obviously don't want to run out of cells, but there's no point in making more than you need. So that your show reel movie stays as flexible as possible, plan on using `duplicateMovieClip()` to add more cells as and when they're needed. All the decision making then goes on within the ActionScript, giving you a much more versatile movie.

Neat, huh? OK, you'll believe it when you see it—it's a trick, obviously.

15. There's just one more thing you need to do before the stage is all set: add an instance of the `reel` movie clip to the main timeline, on the pictures layer. Use the Property inspector to name it `reel_mc` and position it at X: 20, Y: 20:



Movie Clip

reel_mc

Instance of: reel

W: 250.0 X: 20.0

H: 250.0 Y: 20.0

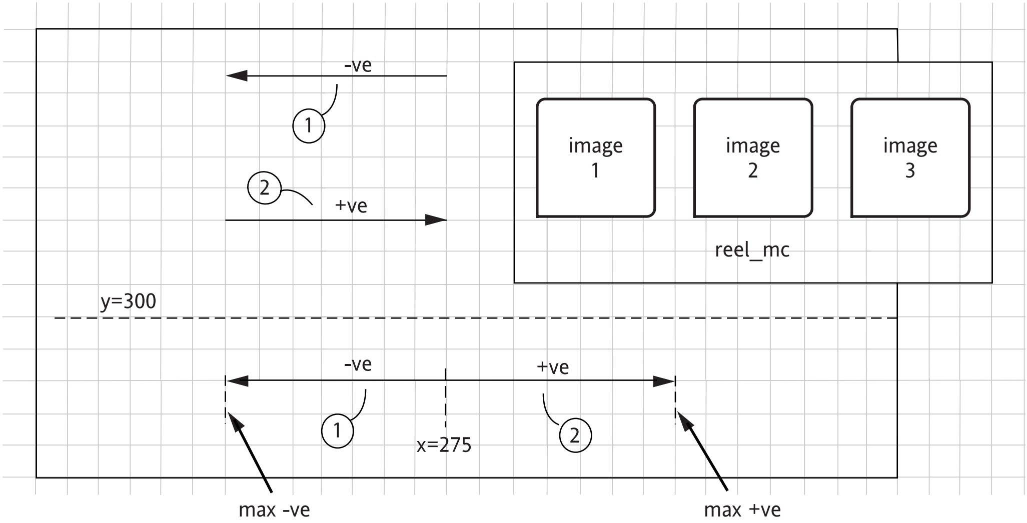
All done!

Now that the stage is ready, it's time to add some ActionScript wiring behind the scenes and make all your movie clips perform some magic. Just like the hangman example you saw in Chapter 5, there's quite a lot of script to get through here, but don't let that put you off! It all breaks down into simple chunks, and you'll take them quite slowly.

> *The code here is fairly involved, so we've created an untyped version because it's easy for beginners to follow. You'll also find a typed version (one that defines all its data types fully) at the end of this exercise.*

Even if you don't understand every single line, you should be able to get an idea of how the movie hangs together (and see how to tweak the important bits so that you can create your own variations). The first of these chunks controls the scrolling. It looks at where the mouse x-coordinate is in relation to a center line and moves the `reel_mc` movie clip instance based on that distance. The direction (left or right) takes care of itself as long as you also consider the *sign* of this distance—whether it's positive or negative. You're not interested in the y-coordinate of the reel, because you're only moving the movie clip from left to right. You're not moving it up and down at all, so it will always stay at the same y-coordinate.

Before you start working on the code for the scroll, you need to make sure that you're very clear on what you need to do. Remember, if it won't work on paper, it won't work when you code it. You need a diagram, so here's a cleaned-up version of a sketch showing how this scroll movement should work:



1. Here's the basic script, which you need to attach to frame 1of the actions layer:

```
_root.onEnterFrame = function() {
  reelSpeed = (_xmouse-center)/10;
  reel_mc._x += reelSpeed;
};
center = Stage.width/2;
```

   Like we said before, it's not dissimilar to the stuff you did back in Chapter 1.

According to this diagram, you need to relate the distance between the mouse and the centerline to the speed at which the reel scrolls. In other words, you want the reel to scroll faster the farther away from the centerline the mouse gets. You've already defined `center` as the "halfway across the stage" value (the x = 275 line in the diagram, which assumes a default stage size of 550 × 400).

```
reelSpeed = (_xmouse-center);
```

You use reelSpeed to tell Flash how many pixels to move reel_mc across the screen every frame. Except, in the finished FLA, it's actually this:

```
reelSpeed = (_xmouse-center)/10;
```

The original line made it scroll too fast, so we just divided it by 10 (after trying a few values to see what gave a good movement) to make it scroll at a more sensible speed. Try running the movie now, and you'll see an instant result: the one-cell show reel moves from left to right under mouse control, just as you want it to.

When the mouse pointer is to the left of the centerline (case 1 in the sketch), _xmouse-center will be negative, giving you a negative speed and reel_mc moving farther to the left with every new frame. If the mouse pointer is to the right of the centerline (as in case 2 in the sketch), you'll get a positive speed and the reel scrolls to the right.

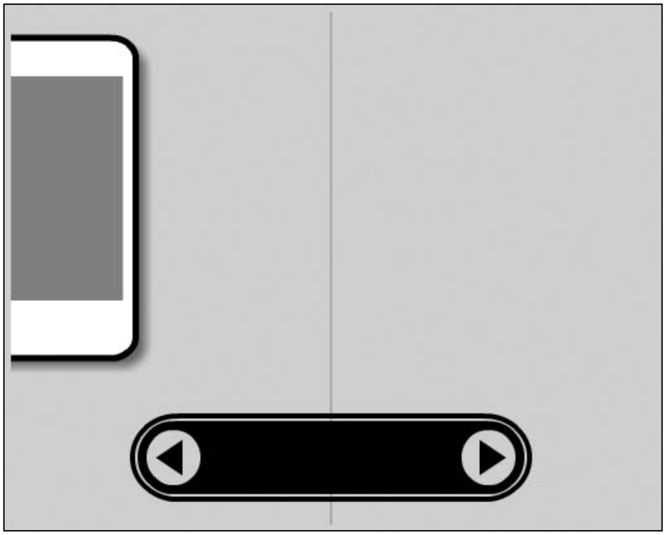
2. There's still a problem, though. The reel doesn't stop when it gets to the edge, so it's quite easy to lose it off the side of the stage! To fix this, you need to put some limits on where reel_mc can move. Let's set up a stop value at either end. You can stop changing reel_mc._x if it ever tries going beyond these values:

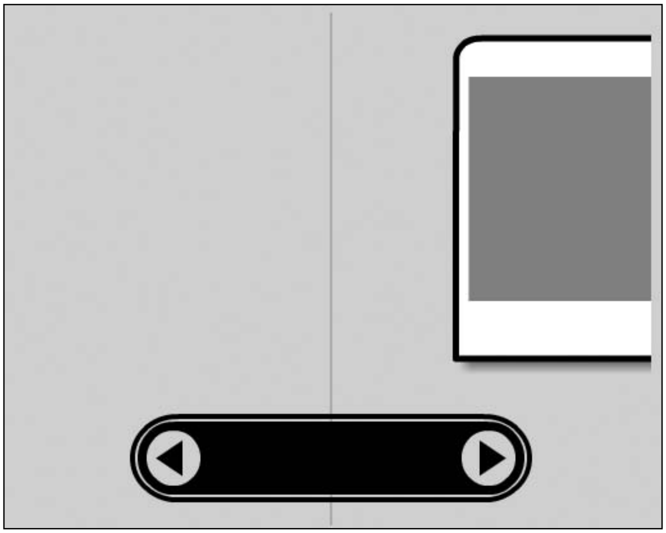
```
// Handler to move show reel left and right across the stage
_root.onEnterFrame = function() {
  reelSpeed = (_xmouse-center)/10;
  reel_mc._x += reelSpeed;
  // Apply limits to reel position
  if (reel_mc._x < LEFTSTOP) {
    reel_mc._x = LEFTSTOP;
  } else if (reel_mc._x > RIGHTSTOP) {
    reel_mc._x = RIGHTSTOP;
  }
};
cellWidth = reel_mc.cell_mc._width;
center = Stage.width/2;
LEFTSTOP = 0-cellWidth/2;
RIGHTSTOP = Stage.width-cellWidth/2;
```

Now, if the reel moves too far in one direction or the other, you stop it dead in its tracks.

The LEFTSTOP and RIGHTSTOP values are set up so that the registration point of reel_mc will go past them when half of reel_mc is offscreen in either direction. The half of a width comes from cellWidth, which contains the width value of the cell. The new if statement detects either of these conditions and stops the clip from moving any farther in the same direction.

The maximum that you can now go in either direction is shown in the following images, which show the SWF as viewed in a browser, allowing us to see the edges.





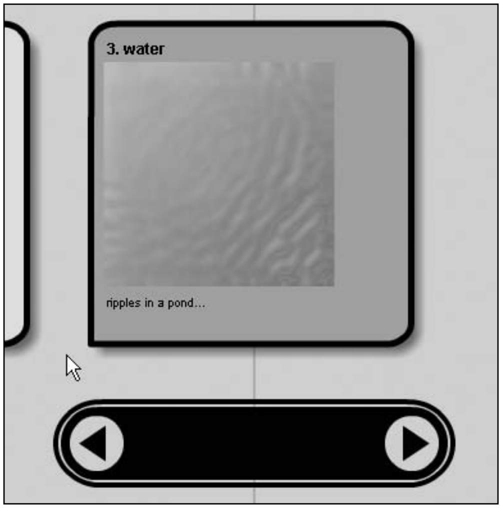
Note that some of the variables are in uppercase letters. It's customary to denote constants in this way, which is what the left and right stops are. Once they're defined, you never change them.

That's fine as long as you have only one cell in the show reel, but what about when you add some more? With the stops as they are, you can't scroll anything off the left side of the stage, so you're never going to see a third or fourth cell, let alone a fifth, a sixth, or a seventh.

3. It looks like you need to rethink the stops. Ideally, you should be able to scroll far enough to see *any* one of the cells in the middle of the stage, no matter how many you have. The following should do the trick (change the last two lines in the listing so far):

```
...
...
cellWidth = reel_mc.cell_mc._width;
center = Stage.width/2;
LEFTSTOP = center - reel_mc._width + cellWidth/2;
RIGHTSTOP = center - cellWidth/2;
```

The stops now ensure that the middle of the cell at either end of the show reel doesn't go past the centerline. In the following screenshot from the finished application, you can see this. The cell is at the farthest right it can go, halfway across the centerline.



If you run the movie now, you'll see that you've killed the scrolling movement! The single cell is trapped in the middle of the stage; the new stops mean that it can't move either left *or* right. You can prove that this is the case (and not that you get no movement because you've somehow broken the animation) by changing either LEFTSTOP or RIGHTSTOP to what it was previously.

Believe it or not, this is a *good* thing. The same cell occupies the left *and* right ends of your show reel, so the fact that it's confined to the center of the stage proves that your stops have worked. Of course, they'll be a lot more useful when you've added in a few more cells!

At this stage, you have a perfectly good scroller. You just need to add in some pictures and text, and take it away. You could add some more cell instances, import some graphics, add some static text fields, and so on. Great.

However, you're aiming a bit higher than that. Now that you have some ActionScript power, you'll make it really work for you. Rather than hard-wiring cells and images into your movie, you're going to get ActionScript to figure out what you want and do it for you.

Next chunk, then: you'll write an ActionScript function that adds new cells.

4. You know how to write functions, and you know how to make duplicate clips, so let's put the two together. The function will define two arguments: num (telling it *which* cell you want to set up) and details (telling it *how* you want the cell set up). Here's the basic function declaration:

   ```
   function newCell (num, details) {
     originalClip = _root.reel_mc.cell_mc;
     newClip = originalClip.duplicateMovieClip("cell"+num, num);
     newClip._x = num*300;
   }
   ```

   You start off by using originalClip to store a reference to the original cell instance, which is tucked away inside reel_mc in the root timeline. This just means you can now use the originalClip container in your script in place of the bulky _root.reel_mc.cell_mc, which is a bit of a mouthful in anyone's book!

   Next, you use duplicateMovieClip to make a copy of the original cell. The new movie clip is called cell1, cell2, or whatever (depending on what value num has), and you store a reference to *that* in a container called newClip.

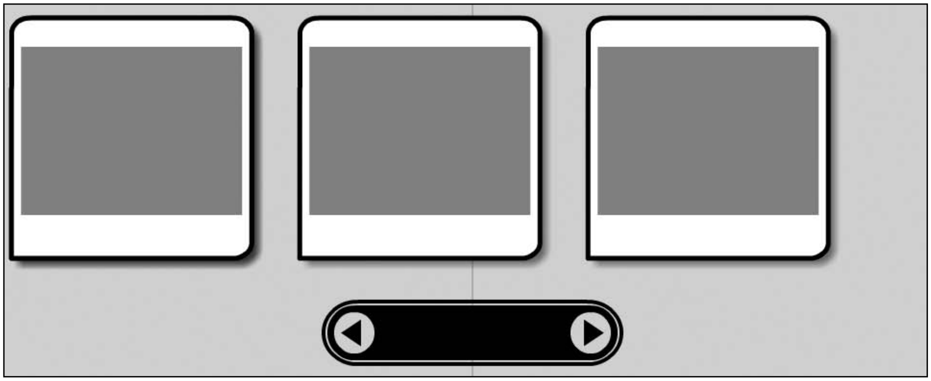
   Now you tell Flash to set the _x property of newClip to num*300. newClip doesn't actually have an _x property, but it's pointing to something that does: the new movie clip. The end result is that your first cell (num=0) gets placed at X:0, while the second cell (num=1) goes to X:300, the third (num=2) goes to X:600, and so on.

> *We chose the number 300 here because for the container clips (260 pixels wide, if we include the additional 10 pixels of the drop shadow), this leaves a space between containers of 40 pixels, which seems to look about right. You'll find that many of the numbers you end up using in a Flash movie are based on what looks right or moves the way you want, rather than the result of some long-winded calculation.*

5. Now just add a couple of newCell function calls just below the newCell function:

   ```
   function newCell (num, details) {
     originalClip = _root.reel_mc.cell_mc;
     newClip = originalClip.duplicateMovieClip("cell"+num, num);
     newClip._x = num*300;
   }
   newCell(0, "");
   newCell(1, "");
   newCell(2, "");
   ```

Run the movie and you'll see you now have a grand total of three cells that scroll back and forth across the screen (shown below). All your new movie clips are created *inside* `reel_mc`, so the coordinates are relative to that. Your original script moves `reel_mc`, so all the cells move with it.



6. So you've got lots of nice blank cells on the stage. They scroll OK, but they aren't much to look at. You need to pass your `newCell` function some more information—a name, for example. There's a new line in the function that sets value of the `name` text field. You use the `details` argument to pull strings out of the function calls:

```
function newCell (num, details) {
  originalClip = _root.reel_mc.cell_mc;
  newClip = originalClip.duplicateMovieClip("cell"+num, num);
  newClip._x = num*300;
  newClip.name_txt.text = num+1 + ". " + details;
}
newCell(0, "Kristian");
newCell(1, "Sham");
newCell(2, "David");
```

7. Alternatively, you could use `details` to pass information about what color you want to make the new cell's background. Here's how you might do that:

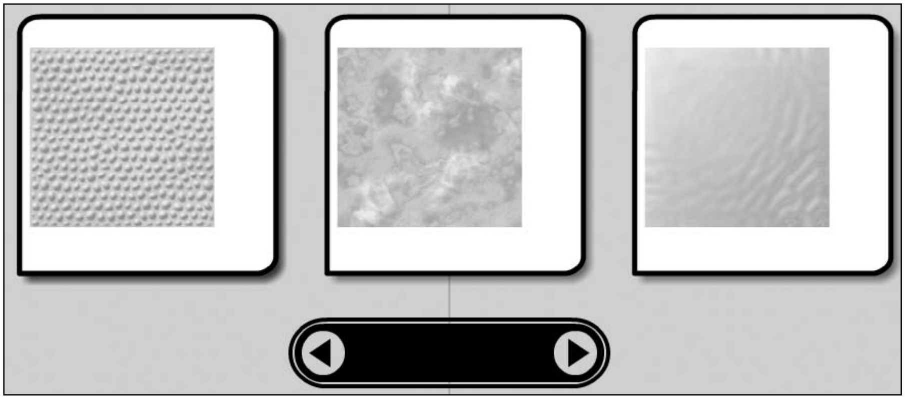
```
function newCell (num, details) {
  originalClip = _root.reel_mc.cell_mc;
  newClip = originalClip.duplicateMovieClip("cell"+num, num);
  newClip._x = num*300;
  // Set color of cell background
  myColor = new Color(newClip.bgnd_mc);
  myColor.setRGB(details);
}
newCell(0, 0xFF9999);
newCell(1, 0x99FF99);
newCell(2, 0x9999FF);
```

This time, you use the `Color` object to set the color of `bgnd_mc` inside each new clip.

8. Most important of all, you want to tell Flash which images to use in each cell. There's a useful action called loadMovie() that you can use to call up image files, but it still needs to know which file you want!

```
function newCell (num, details) {
  originalClip = _root.reel_mc.cell_mc;
  newClip = originalClip.duplicateMovieClip("cell"+num, num);
  newClip._x = num*300;
  // Set cell content
  content = newClip.placeholder_mc;
  content.loadMovie(details);
}
newCell(0, "cobblestones.jpg");
newCell(1, "planet.jpg");
newCell(2, "water.jpg");
```

This time, the new lines load the image specified by details and use it to replace the placeholder_mc movie clip instance in each new cell:



Of course, you actually want all three of these—and perhaps more! You could give the newCell function a few more arguments, passing picName in one, picColor in the next, and link (to the filename of the image to display) in the next.

However, there's another option: you could send the details argument an object. So far, you've always used simple variables as function parameters, passing strings and numbers to help the function do its job. But there's no reason you can't feed it objects as well—then the function can pull out various property values as and when it needs them. You don't have to deal with lots of parameters, and you can define your object properties wherever you like.

1. Let's start by declaring a constructor function for a `Picture` class of object. Every `Picture` object will have four properties, `name`, `link`, `color`, and `description`, so it will look like this:

```
function Picture (name, link, color, description) {
  this.picName = name;
  this.picLink = link;
  this.picColor = color;
  this.description = description;
}
```

2. Assuming you pass your `newClip` function one of these `Picture` objects via its `details` argument, you can use its properties like this:

```
// Define function for creating
// cells in the show reel
function newCell (num, details) {
  originalClip = _root.reel_mc.cell_mc;
  newClip = originalClip.duplicateMovieClip("cell"+num, num);
  newClip._x = num*300;
  newClip.name_txt.text = num+1+". "+ details.picName;
  newClip.description_txt.text = details.description;
  picContent = newClip.placeholder_mc;
  picContent.loadMovie(details.picLink);
  myColor = new Color(newClip.bgnd_mc);
  myColor.setRGB(details.picColor);
}
```

Here you combine all the earlier cell customizations (plus a new one that sets the description text) into one version, which uses the four properties of the `Picture` object referred to by `details`. Of course, this relies on you calling `newCell` with proper parameters.

3. Here's the last piece in the puzzle:

```
pics = new Array();
// Set up picture objects
pics[0] = new Picture("cobblestones", "cobblestones.jpg", 0xFF9999,➥
 "from an English country lane...");
pics[1] = new Picture("planet", "planet.jpg", 0x99FF99, ➥
 "the view from 400km up...");
pics[2] = new Picture("water", "water.jpg", 0x9999FF,➥
 "ripples in a pond...");
newCell(0, pic0);
newCell(1, pic1);
newCell(2, pic2);
```

You create three objects, specifying properties as you go, and you create the three cells quite separately. In fact, this would be even tidier if each Picture object was one element in an array. Then you can simply loop through all the elements and create a new cell for each before hiding the original cell_mc (which has served its purpose):

```
pics = new Array();
// Set up picture objects
pics[0] = new Picture("cobblestones", "cobblestones.jpg", 0xFF9999,➥
 "from an English country lane...");
pics[1] = new Picture("planet", "planet.jpg", 0x99FF99,➥
 "the view from 400km up...");
pics[2] = new Picture("water", "water.jpg", 0x9999FF,➥
 "ripples in a pond...");
// Loop through array of picture objects,
// creating a new cell for each
for (var i = 0; i<pics.length; i++) {
  newCell(i, pics[i]);
}
// Hide the original cell movie clip after duplicating it
reel_mc.cell_mc._visible = false;
```

Phew! I'm sure you'll be very pleased to hear that the script is done! Try the movie out again, and you should get the full effect of a scrolling picture gallery.

So, was all that hard work *really* worth it? You seem to have spent quite a lot of effort getting nowhere very quickly. The scrolling effect is pretty straightforward, and most of the script seems to just be mucking about with data behind the scenes. That's as it may be, but thanks to all that mucking about, you have a show reel whose contents are completely defined by four lines of script, namely the objects stored in the pics array. Here's that script in full:

```
// Handler to move show reel left and right across the stage
_root.onEnterFrame = function() {
  reelSpeed = (_xmouse-center)/10;
  reel_mc._x += reelSpeed;
  // Apply limits to reel position
  if (reel_mc._x<LEFTSTOP) {
    reel_mc._x = LEFTSTOP;
  } else if (reel_mc._x>RIGHTSTOP) {
    reel_mc._x = RIGHTSTOP;
  }
};
// Define constructor function for Picture objects
function Picture (picName, link, picColor, description) {
  this.picName = picName;
  this.picLink = link;
  this.picColor = picColor;
  this.description = description;
}
```

```
// Define function for creating cells in the show reel
function newCell (num, details) {
  // Create a new cell
  originalClip = _root.reel_mc.cell_mc;
  newClip = originalClip.duplicateMovieClip("cell"+num, num);
  // Set cell position and text fields
  newClip._x = num*300;
  newClip.name_txt.text = num+1+". "+ details.picName;
  newClip.description_txt.text = details.description;
  // Set cell content
  picContent = newClip.placeholder_mc;
  picContent.loadMovie(details.picLink);
  // Set color of cell background
  myColor = new Color(newClip.bgnd_mc);
  myColor.setRGB(details.picColor);
}
// Initialize useful variables and picture array
cellWidth = reel_mc.cell_mc._width;
center = Stage.width/2;
pics = new Array();
//
// Set up picture objects
pics[0] = new Picture("cobblestones", "cobblestones.jpg", 0xFF9999,➥
 "from an English country lane...");
pics[1] = new Picture("planet", "planet.jpg", 0x99FF99,➥
 "the view from 400km up...");
pics[2] = new Picture("water", "water.jpg", 0x9999FF,➥
 "ripples in a pond...");
// Loop through array of picture objects, creating a new cell for each
for (var i = 0; i<pics.length; i++) {
  newCell(i, pics[i]);
}
// Hide the original cell movie clip after duplicating it
reel_mc.cell_mc._visible = false;
// Define constants (these cannot be defined until the pictures
// have been placed onscreen)
LEFTSTOP = center-reel_mc._width+ cellWidth/2;
RIGHTSTOP = center-cellWidth/2;
```

Anytime you want to change the content or appearance of one of your cells (or even add a completely new one), you can simply change (or add) an element in the `pics` *array.*

Again, don't panic if you don't understand every line in here—that's not the point. What you should be starting to appreciate is that it pays to *structure* your scripts. Break them down into task-focused chunks, and the ease with which you can modify them later on should more than make up for the extra effort involved in writing them in the first place.

You should also add as many comments as you need (and more than you need, because you'll most likely forget what's going on in your code six months down the road!). We haven't added that many comments because we're explaining what's happening in the text of this book (plus the fact that you have to type all the code in, so fewer lines are better), but for your own code, go to town with comments.

Oh, and one final thing for the fast-trackers: the same script, this time with strict typing. You'll see it's essentially the same, but it looks a lot longer and more complicated (and we didn't want to scare anyone off!):

```
// Handler to move show reel left and
// right across the stage
_root.onEnterFrame = function() {
  var reelSpeed:Number = (_xmouse-center)/10;
  reel_mc._x += reelSpeed;
  // Apply limits to reel position
  if (reel_mc._x < LEFTSTOP) {
    reel_mc._x = LEFTSTOP;
  } else if (reel_mc._x > RIGHTSTOP) {
    reel_mc._x = RIGHTSTOP;
  }
};
// Define constructor function for
// Picture objects
function Picture(picName:String, link:String, picColor:Number,➥
 description:String):Void {
  this.picName = picName;
  this.picLink = link;
  this.picColor = picColor;
  this.description = description;
}
// Define function for creating cells in the show reel
function newCell(num:Number, details:Object):Void {
  var originalClip:MovieClip;
  var newClip:MovieClip;
  var picContent:MovieClip;
  var myColor:Color;
  // Create a new cell
  originalClip = _root.reel_mc.cell_mc;
  newClip = originalClip.duplicateMovieClip("cell"+num, num);
  // Set cell position and text fields
  newClip._x = num*300;
  newClip.name_txt.text = num+1+". "+ details.picName;
  newClip.description_txt.text = details.description;
  // Set cell content
  picContent = newClip.placeholder_mc;
  picContent.loadMovie(details.picLink);
  // Set color of cell background
  myColor = new Color(newClip.bgnd_mc);
  myColor.setRGB(details.picColor);
}
// Initialize useful variables and picture array
var cellWidth:Number = reel_mc.cell_mc._width;
var center:Number = Stage.width/2;
var pics:Array = new Array();
//
// Set up picture objects
pics[0] = new picture("cobblestones", "cobblestones.jpg", 0xFF9999,➥
 "from an English country lane...");
pics[1] = new picture("planet", "planet.jpg", 0x99FF99,➥
 "the view from 400km up...");
pics[2] = new picture("water", "water.jpg", 0x9999FF, ➥
 "ripples in a pond...");
// Loop through array of picture objects, creating a new cell for each
for (var i = 0; i<pics.length; i++) {
  newCell(i, pics[i]);
}
// Hide the original cell movie clip after duplicating it
reel_mc.cell_mc._visible = false;
// Define constants (these cannot be defined until the pictures
// have been placed onscreen)
var LEFTSTOP:Number = center-reel_mc._width+cellWidth/2;
var RIGHTSTOP:Number = center-cellWidth/2;
```

You could have used Function *as the type for* details *in the function definition for* details *in* newCell. *Although* details *is an object, it's actually defined via a constructor function, and you're passing the function as an argument (maybe we'll leave that for another day!).*

```
function newCell(num:Number, details:Function):Void {
```

Book project: Initializing the code

So far we've concentrated on the graphics and design of our site. We haven't really done any coding, but we've been building up the main nonscript parts of the site.

In this section, you will do two things:

- First off, you'll be doing very little with the graphics, timeline, or stage, so you'll customize the Flash environment for some serious scripting.
- Next, you'll initialize your site. This consists of setting the variables that your site will use to their initial values.

The final FLA for this section is included as futuremedia_initialize.fla. The starting point is the file you worked on in Chapter 6, futuremedia_setup3.fla.

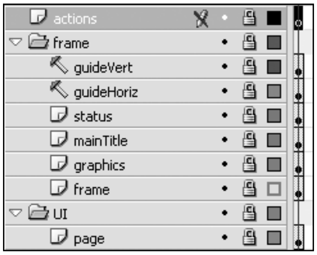
Getting yourself comfortable

Although you may be used to designing site user interfaces with heaving libraries full of lots of symbols, the Futuremedia user interface actually now has everything it needs. All you need to do is add the content, but you will do that once the UI itself is completed. Since you will not be touching either the timeline or stage for some time, you may as well rearrange the furniture for the job of coding.

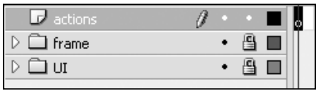
> *Flash 8 Professional has a special file type called the AS (ActionScript) file. This is a code-only format, and it has an associated dedicated coding screen that consists of a full-screen ActionScript editor. Although this editor is a better option for creating the code you'll be writing for the Futuremedia site, the AS file format is not designed for use by the ActionScript beginner. We'll look at the AS format later in the book, when you'll be better able to use it.*

The following environment is what we used to develop the site. It was authored on a PC with a 1600 × 1200-sized screen. You may want to make minor changes if you are on a Mac or if you have a different screen resolution, but barring those who are set in their ways, we'd like to forewarn you that you're going to end up with probably the longest ActionScript listing you've ever seen, and YOU will be the one writing it! The days of three-line button scripts and ten-line `onEnterFrame` scripts are well behind you. You'll have developed a 400+ line script by the end of this book, and you need a totally different environment from the one you've been used to for tween-based sites.

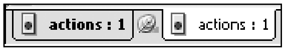
1. At the moment, you'll have a timeline that looks like the following.



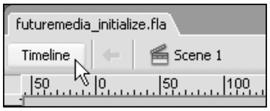
Close the frame and UI folders if yours are still open, and select frame 1 of the actions layer.



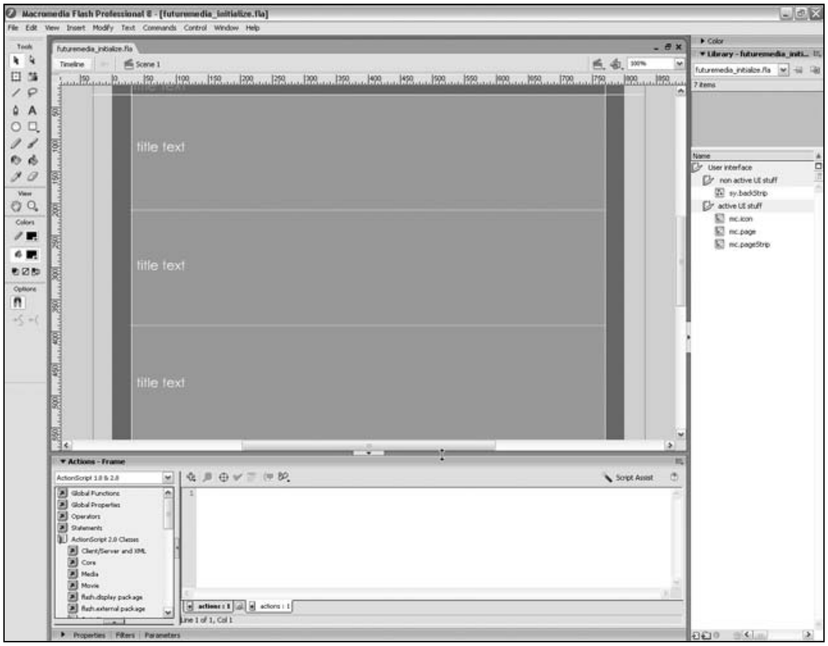
2. So we're all using the same basic authoring environment, select Window ➤ Workspace Layout ➤ Default. Minimize the Property inspector. Maximize the Actions panel, click the pin icon, pinning the focus of the Actions panel to frame 1 of layer actions.

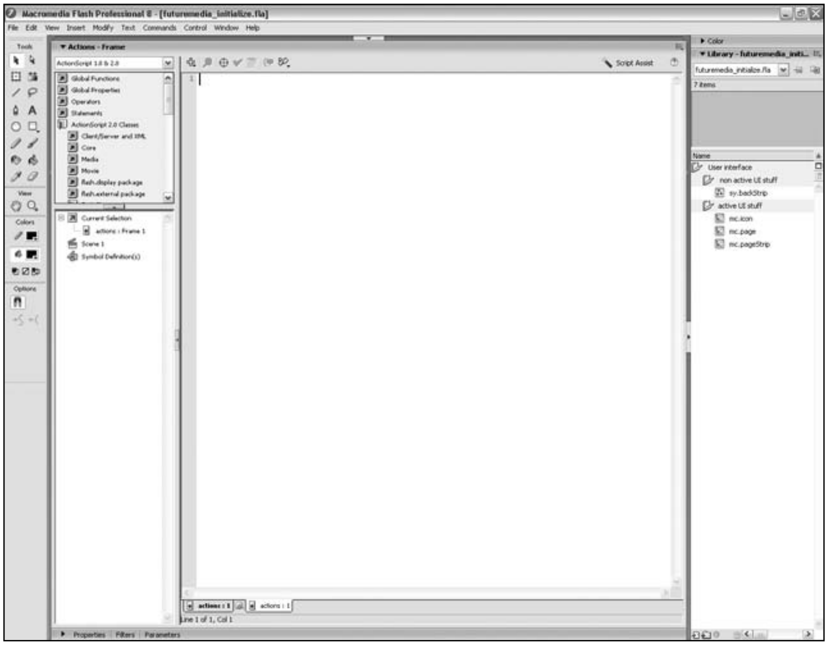


3. Minimize the Timeline panel.



4. By pulling the top edge of it upward, extend the Actions panel to cover the stage area.





You've replaced the big wide stage area with a big wide scripting area!

Initializing the site variables

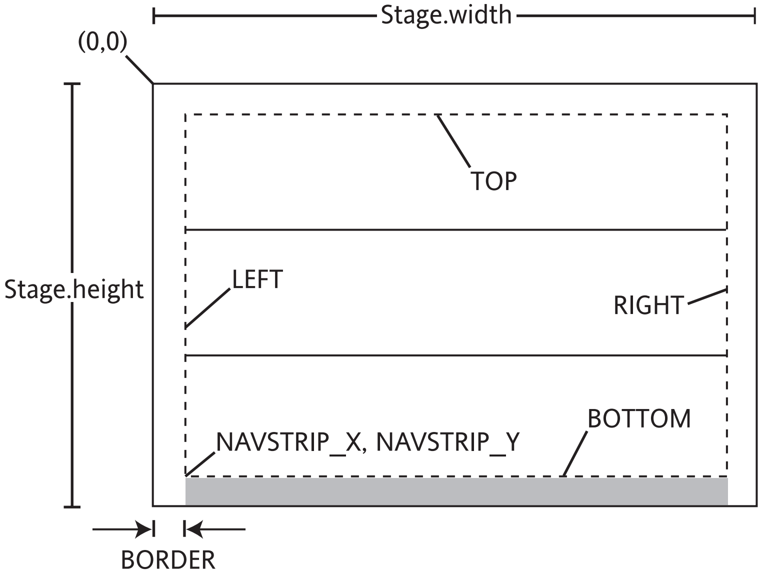
There are two lines of thought regarding initializing variables:

- The first involves initializing variables as and when they are needed. It is part of a fast way of coding commonly called "hacking it." It is fast because you don't have to think too far ahead. Obviously, this lack of foresight can come back to haunt you later!
- The second involves sitting back and thinking about what data your code will actually need in general terms, and then adding the data as variables. It involves drawing sketches, marking out your major points of reference, and converting them into variables and other data. Although you won't get all your variables defined, you will get the major ones, and that will help when you come to define your code.

As well as being better practice, the second approach has another more subtle advantage: it makes you see the process of coding in the proper perspective.

Although most programming books concentrate on code, the professional programmer knows that the thing that drives any application is not code—it is data.

Before you can initialize your variables, you need to think about which variables you actually need. This application is visual, so your "create a user interface" problem is all about positioning on the stage. Time for another sketch:



There are a number of points on the screen that your scripts will need to know about:

- The height and width of the stage area, because most of your measurements will be based upon the stage size, `Stage.height` and `Stage.width`.
- The thickness of the border area or "gutter" that encloses the main content, `BORDER`.
- The position of the top, bottom, left, and right edges of the main UI, `TOP`, `BOTTOM`, `LEFT`, and `RIGHT`.
- The position of the back navigation strip, `NAVSTRIP_X` and `NAVSTRIP_Y`, because to place your icons on it, you need to first know where it starts.

Telling ActionScript about the stage

To capture the stage dimensions, you can simply read them from the Document Properties window (Modify ➤ Document). That's cool, but it would be better if you could get the stage dimensions from Flash during runtime, because you may later decide to change the stage size. The `Stage` class allows you to get the stage dimensions directly from the Flash Payer during runtime.

1. Add this in the Script pane of the Actions panel:

```
trace(Stage.width);
trace(Stage.height);
```

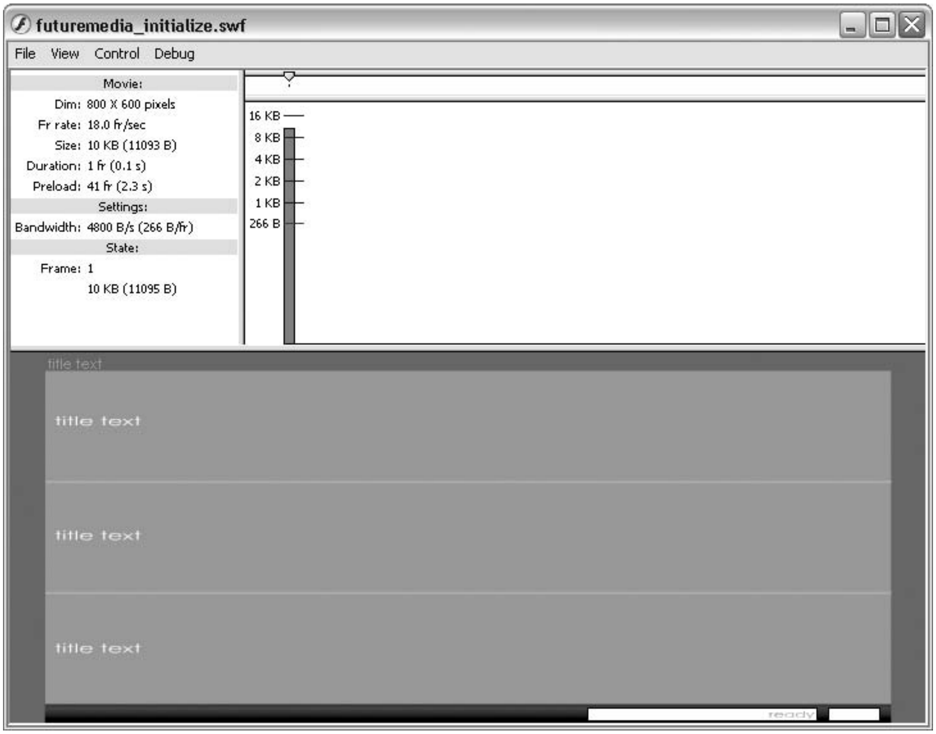
Test the FLA. On a Windows machine, this will *not* give you the values 800 and 600 for width and height. It will give you the height and width of the current window that is showing the SWF. On the Web, this window will be the browser window. To fix this, you have to tell Flash to show the movie in the Exact Fit mode (this is one of the same display modes that you can set in File ➤ Publish Settings ➤ HTML ➤ Scale).

> *On the Mac you should get the correct result of 800 for width and 600 for height every time, as the test movie is always correctly scaled on the Mac.*

2. Since you are an Flash ActionScript Master in the making, you'll make the change via scripting rather than via the Publish Settings. Add the following line as shown:

```
Stage.scaleMode = "exactFit";
trace(Stage.width);
trace(Stage.height);
```

If you test the FLA now, you will find that you see the correct dimensions. However, this is *not* the display mode you want—this is the "scale to fit the current window" mode. While in test mode, press Ctrl+B/Cmd+B to bring up the Bandwidth Profiler to see why. Worse still, make the Bandwidth Profiler bigger (click-drag its lower edge). The Futuremedia site is scaled to fit in its newly resized window, but doesn't maintain its proportions . . . urgh!



Movie:
Dim: 800 X 600 pixels
Fr rate: 18.0 fr/sec
Size: 10 KB (11093 B)
Duration: 1 fr (0.1 s)
Preload: 41 fr (2.3 s)
Settings:
Bandwidth: 4800 B/s (266 B/fr)
State:
Frame: 1
10 KB (11095 B)

The Futuremedia site (like most other Flash sites) is designed to be viewed at a particular size and a particular height:width ratio. Viewing it at any other size may make it look odd and may also affect performance if the user decides to view the site at full screen.

3. Not to worry, if you can change the display mode, you can change it back! Add the following line:

```
Stage.scaleMode = "exactFit";
trace(Stage.width);
trace(Stage.height);
Stage.scaleMode = "noScale";
```

That’s better!

Between the first and last line of this script, Flash will give you stage measurements with respect to the Flash stage (i.e., it will assume an 800 × 600 stage).

If you look at the stage dimensions without the first and last line, Flash will return ***actual pixel dimensions****. For example, if you scaled the entire site by 50% in the publish options (so that, for example, you could view the site on a wireless device with a smaller screen area), Flash would also change the screen dimensions it returned when you looked at* `Stage.width` *and* `Stage.height`.

The way we've done it here means this result doesn't occur—you can scale the site without affecting your code. This gives the site and its associated code a very useful property: it is ***resolution independent****. The code will always see an 800 × 600 screen, irrespective of what is going on outside the Flash Player with regard to scaling (or anything else).*

OK, you now have almost all the values to base the site measurements on except one: the border. It is actually 30 pixels from the stage edge all the way around, and this is how you tell Flash:

```
border = 30;
```

Simple!

4. Delete all the code you have added so far and add the following script:

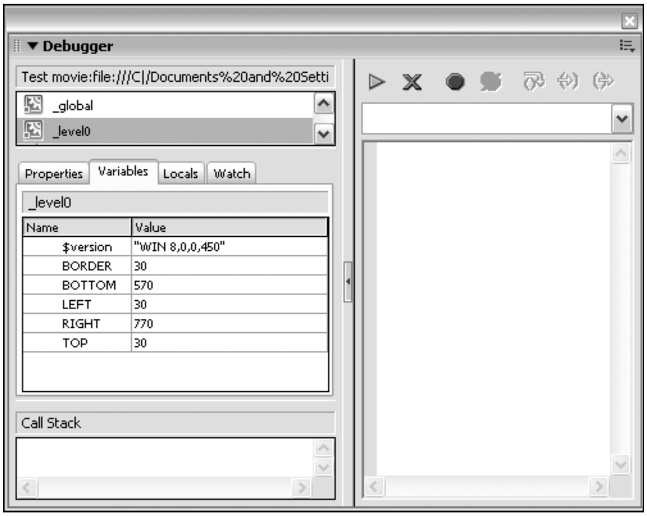
```
// Define screen extents for later use...
Stage.scaleMode = "exactFit";
var BORDER:Number = 30;
var BOTTOM:Number = Stage.height-BORDER;
var TOP:Number = BORDER;
var RIGHT:Number = Stage.width-BORDER;
var LEFT:Number = BORDER;
Stage.scaleMode = "noScale";
```

This defines all your important screen measurements in terms of the three primary measurements: `HEIGHT`, `WIDTH`, and `BORDER`. Have a look at the sketch from a little while back if you're unsure about any of them.

Um . . . why the CAPITAL LETTERS? It is a programming convention to give constant values (i.e., variables that will never change their initial value) a variable name in capital letters and underscores. A variable "my variable" would thus be called `MY_VARIABLE` if it was a constant and `myVariable` if it was not.

We have not included the two measurements that show where the back strip is yet. You will do that separately later—it will require a major bit of code in its own right.

You can test that the code is working so far by using the Debug Movie mode (Control ➤ Debug Movie or Ctrl+Shift+Enter/Cmd+Shift+Enter). Click the green "arrow" icon at the top right of the Debugger pane to start the debug session. If you highlight _level0 in the top-left pane and then select the Variables tab you will see your variables appear with their current values:



Test movie:file:///C|/Documents%20and%20Setti

_level0

Name	Value
$version	"WIN 8,0,0,450"
BORDER	30
BOTTOM	570
LEFT	30
RIGHT	770
TOP	30

Sanity check

*A **sanity check** is something that you will see often in the book as you go through the project code. It describes those moments of pure frustration and gradual loss of sanity that can occur when learning programming from books, just as you are now, and the book shows something nothing like the nonworking rubbish experienced by the reader (yep, it is not just you). The sanity checks are points where we will present some additional information to help you out of the mess. These will usually be a subtle error that you might miss or a general cross-check to make sure we are moving forward with the same code.*

If you feel you are about to kick a wall or click your mouse really hard, you may find it better to scan ahead to the nearest sanity check first!

You're using ActionScript 2.0 code for this site. The code will not work (and will give no indication why) if your ActionScript version publish options are set to ActionScript 1.0 and you will see no variables at all in the debugger. By default, a movie created in Flash 8 will be set to ActionScript 2.0, but there might be a time in your mammoth Flash career where nothing happens and, thanks to us, now you know where to look!

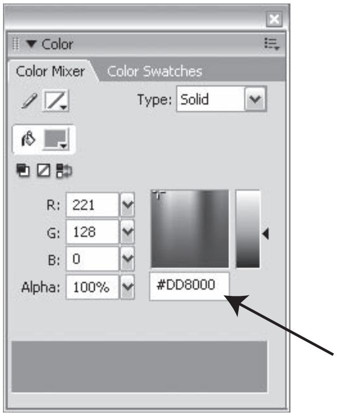
If you are seeing no variables at all, go to File ➤ Publish Settings ➤ Flash tab, where you should see the following:

Version: Flash Player 8

ActionScript version: ActionScript 2.0

Setting up your tricolor colors

The next thing you want to add is the initial colors of the strips. We've chosen some colors for the site: #993333, #119922, and #005577. You can get an early look at these colors by entering them into the Color Mixer panel. You can change them more to your liking if you violently disagree with these choices later, but leave them as they are for now so that we all develop the same site.



Type: Solid

R: 221

G: 128

B: 0

Alpha: 100%

#DD8000

Enter a hex value here to view it.

1. The site will use these colors to tint the first set of strips and will base all further page colors on them. They therefore define the colors of *all* the backgrounds in the site. Add the following script to the bottom of what you have so far:

```
// Define initial screen colors (further colors will
// use these colors as their seed)
var color0:Number = 0xDD8000;
var color1:Number = 0xAACC00;
var color2:Number = 0x0080CC;
```

color0 to color2 are the three variables that hold your Hex color values. The 0x at the beginning of each of the colors means that it is in hexadecimal. If you test your code so far, you will see decimal values for the color variables, and they will be 14516224, 11193344, and 32972.

2. Add your NAVSTRIP_X and NAVSTRIP_Y measurements now. Add the following code to the end of the script so far:

```
// Define navigation strip variables...
var NAVSTRIP_X:Number = BORDER;
var NAVSTRIP_Y:Number = BOTTOM;
```

Note again that you change your variables to CAPITAL_FORMAT when you implement these variables in code. This is to denote constants.

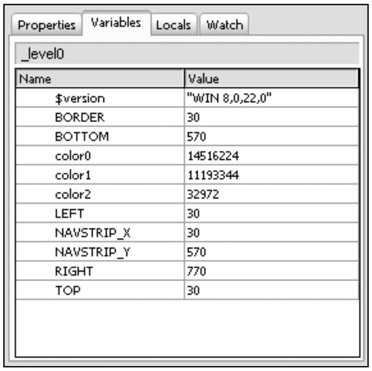
The color variables here have been left as "normal" variables in lowercase, not CAPITALIZED constants. This is just in case you change the value of these variables later on.

OK, if we're honest, we aren't going to change the colors in the Futuremedia website dynamically—we like the colors that we've chosen! However, you might want to alter the case study website to make a website where users can pick their own color scheme. Alternatively, you could have a website where the colors shift through the spectrum. Cool!

You have now initialized all the main variables that define your graphic UI. Flash now knows where everything is.

Sanity check

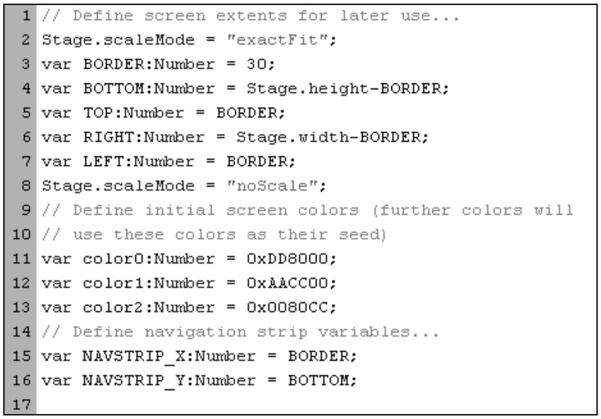
You should see the following variable names and variable values in the debugger. If you are seeing any showing up as undefined, *you need to cross-check with the following listings*. If you are seeing the same thing as in our debugger, you can skip checking against the listings—congratulations, you've correctly initialized the site! Feel free to award yourself a coffee/tea/beer/mineral water break.



Name	Value
$version	"WIN 8,0,22,0"
BORDER	30
BOTTOM	570
color0	14516224
color1	11193344
color2	32972
LEFT	30
NAVSTRIP_X	30
NAVSTRIP_Y	570
RIGHT	770
TOP	30

> *You need to check that all your variables are the same value* and *name as the ones shown in the debugger—ActionScript 2.0 is case sensitive and getting* either *the name or value incorrect now will cause problems later, particularly because the whole site depends on these being right.*

Your onscreen code listing should look like this:



```
// Define screen extents for later use...
Stage.scaleMode = "exactFit";
var BORDER:Number = 30;
var BOTTOM:Number = Stage.height-BORDER;
var TOP:Number = BORDER;
var RIGHT:Number = Stage.width-BORDER;
var LEFT:Number = BORDER;
Stage.scaleMode = "noScale";
// Define initial screen colors (further colors will
// use these colors as their seed)
var color0:Number = 0xDD8000;
var color1:Number = 0xAACC00;
var color2:Number = 0x0080CC;
// Define navigation strip variables...
var NAVSTRIP_X:Number = BORDER;
var NAVSTRIP_Y:Number = BOTTOM;
```

As plain text, it should read like this:

```
// Define screen extents for later use...
Stage.scaleMode = "exactFit";
var BORDER:Number = 30;
var BOTTOM:Number = Stage.height-BORDER;
var TOP:Number = BORDER;
var RIGHT:Number = Stage.width-BORDER;
var LEFT:Number = BORDER;
Stage.scaleMode = "noScale";
// Define initial screen colors (further colors will
// use these colors as their seed)
var color0:Number = 0xDD8000;
var color1:Number = 0xAACC00;
var color2:Number = 0x0080CC;
// Define navigation strip variables...
var NAVSTRIP_X:Number = BORDER;
var NAVSTRIP_Y:Number = BOTTOM;
```

Final words

OK, so you've probably already noticed that each of these book project sections is a passion play, complete with moral at the end.

The aim here was to show the way our code is really nothing more than an alternative expression of the graphic environment that we have built up in the previous sections and, as designers, we are well versed with graphics. Code is not really such a big deal after all.

The code in this chapter simply turns the graphic environment into a set of numbers, and you may have noticed no real "jolt" as you moved from graphic building to code writing. You are still using the same sketches to work out what you need to do; it's just that you're expressing the graphic sketches as code rather than pixels on the stage.

If you're one of those select readers that felt a completely jolt-free transition, there is a very good reason for this. It is because, despite the fact that code may have scared you in the past, in reality you see no difference between graphics and code. You're an ActionScript guru in the making!

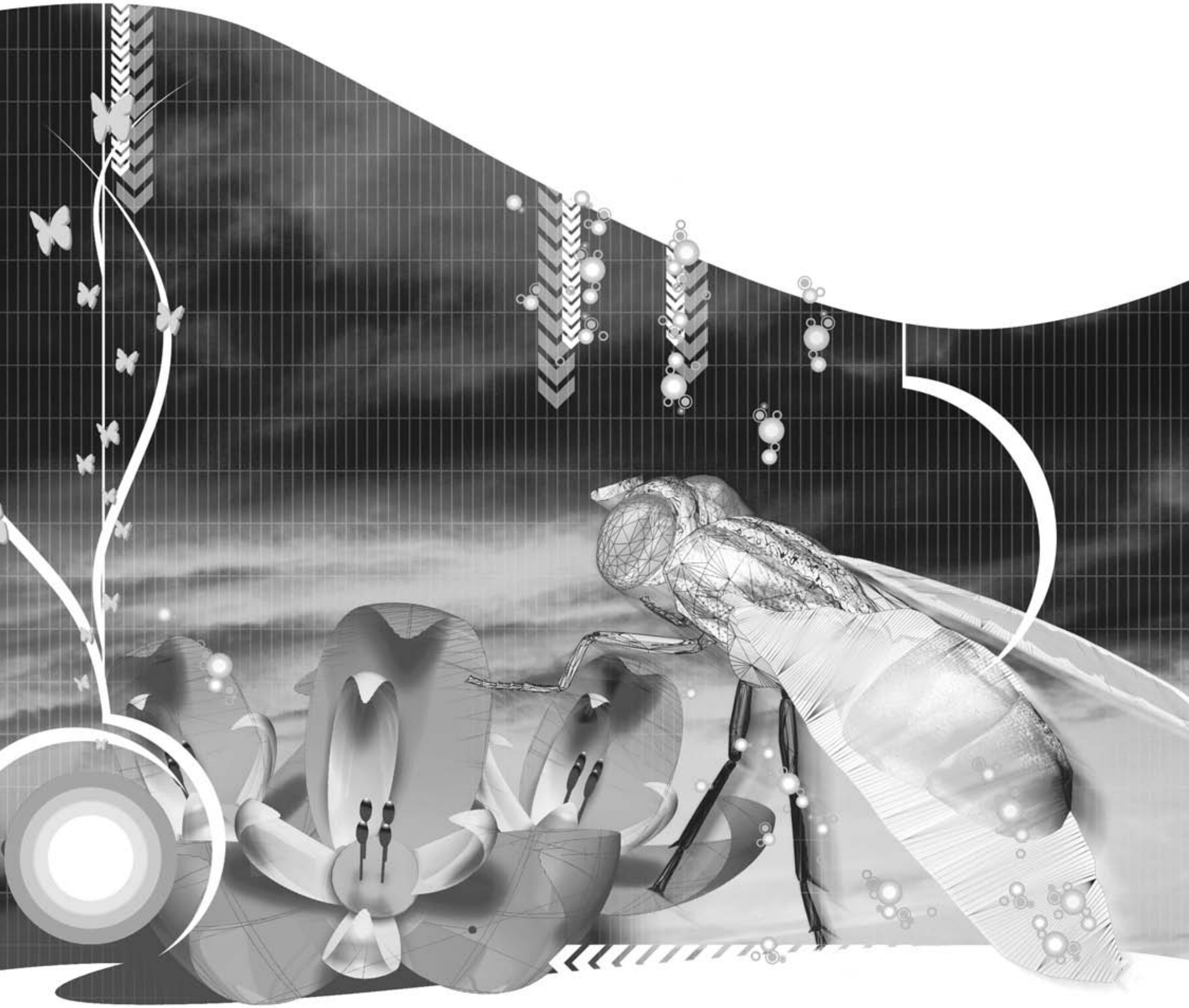
Summary

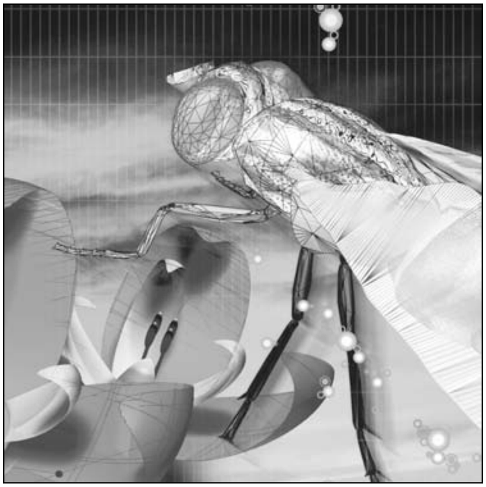
So, what have you learned in this chapter? You've been introduced to objects, which are merely ways of grouping related variables, constants, and functions so that they more closely resemble actual things that you might want to represent in your Flash movies.

As they represent real-life things, objects have state and behavior (they not only *are* things, but they *do* things as well). In ActionScript, objects are described by their **properties** with their actions described by **methods**.

ActionScript has some built-in objects, which represent the sorts of standard elements that you'd want to include in your movies (such as numbers and strings), but you can also create your own objects to represent anything you wish. You've learned the difference between **classes** (which are general definitions of an object, such as *Car* or *Tree*) and **instances** (which are specific objects, such as *Ferrari* or *Oak*).

In the next chapter, you're going to tie the general concept of objects within code to something you've been using in Flash since before you'd heard of ActionScript—the movie clip—and you're going to see that you've been using objects all along, without even realizing it.





Chapter 8

OBJECTS ON THE STAGE

What we'll cover in this chapter:

- Understanding how items on the stage correspond to objects in scripts
- Structuring scripts to take advantage of nested movie-clip timelines
- Using the properties of graphical objects to build advanced interactivity
- Understanding how bitmap caching can improve the performance and frame rate of Flash movies

OK, let's not waste any time here. There's one main idea to wrap your brain around in this chapter: the movie clips, buttons, and text fields you've been using since way back when *are all objects*.

Not only does this fact make it a lot easier to explain a whole variety of ActionScript syntax, but it also opens up a lot of possibilities for what you can do to the things you see on the stage in the course of a movie.

The clues were all there, but kudos to you if you spotted this fact already! If not, don't worry, because you're about to take a fresh look at these old friends.

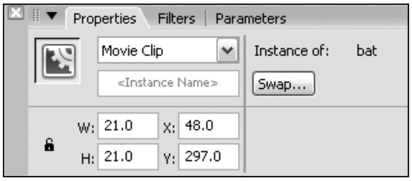
You already know that objects are useful for grouping methods and properties that have some sort of common theme. But what do the pure ActionScript objects (like `Array` and `String`) have to do with movie clips and things? Nothing at all, except that they have the same basic internal structure. They're all objects (or instances, which means the same thing), and they all contain properties and methods.

Words like "instance" and "property" have cropped up a few times when we've discussed items on the stage and, although movie clips have a visual appearance and a timeline, they're really just the same as data-only objects such as numbers and strings.

The reason you can see a movie clip is that Flash represents some of the properties of the movie clip class visually in the Flash authoring environment. This appearance, however, is still fundamentally tied to the same things: methods and properties.

Movie clips and buttons as objects

As you know, to control an item on the stage, you need to give it an instance name, and you can do this via the Property inspector. For example, say you have a movie clip called `bat` in the Library. When you drag an instance of `bat` onto the stage and select it, the Property inspector will look like this:


Movie Clip
Instance of: bat
<Instance Name>
W: 21.0 X: 48.0
H: 21.0 Y: 297.0

From here, you can set the instance name of this particular `bat` symbol and also tweak its position, width, height, and even color. Any changes you make will affect only this particular instance (which you'll call `bat_mc`), and any other `bat` instances you decide to put on the stage will have their own quite separate properties.

Once you've assigned an instance name to your movie clip instance, you can control these properties (and many more) from ActionScript. What's more, you do it in the same way as you controlled object properties in the last chapter. For example, you can set the x-coordinate like this:

```
bat_mc._x = 200;
```

So to control the bat's x-coordinate, you set a property called `_x` for the `bat_mc` movie clip instance. You did something similar when you used the `_visible` property to hide instances and used the `text` property of a dynamic text field to control what text appeared in it.

> *The reason some properties start with an underscore (_) and others don't is due to a historical oddity. In older versions of Flash, all properties started with _, but more recent additions to the language don't.*

You know that there are plenty of things a movie clip can *do*, so you expect to find at least one or two methods. For example, you can stop the `bat_mc` timeline like this:

```
bat_mc.stop();
```

But surely that's just the good old `stop()` action being applied to `bat_mc`? Yes, it is. But think for a moment about *what* an action really is. It's just an instruction to your movie (or to a movie clip) to *do* something. As far as ActionScript is concerned, there's no difference between applying the `stop()` action to a movie clip and calling a movie clip's `stop()` method. They both mean the same thing: stop that movie clip!

> *When you're just starting out with Flash, you can talk about actions or commands and nobody will notice, but using their proper name, methods, will never fail to impress. Knowing how to apply methods in Flash is crucial if you want to make it as a top-class ActionScripter.*

You've already seen a few other movie clip methods, such as `duplicateMovieClip()`, which you can use to make copies of a movie clip instance.

So, each instance (whether it's an input text field, a dynamic text field, a button, or a movie clip) has a set of **properties** that relate to its physical appearance. This is a movie clip, so the properties correspond to things like the clip's position, size, visibility, and total number of frames, whereas the methods do things like play, stop, load, and duplicate it. A button instance has its own particular properties and methods, which (among other things) let you control whether or not it can be clicked. Text fields also have their own properties and methods, the majority of which determine how the text will appear on the screen.

> *If you look in the* Movie *book of the* Actions *toolbox (*ActionScript 2.0 ➤ Movie*), you'll see that the classes (*`MovieClip`*,* `Button`*, and* `TextField`*) behind each of these instances also contain lists of event handlers. Yes, the events are part of the objects too—is anything safe? We won't talk about how they fit into the world of objects, since that's pretty advanced stuff that you can safely get along without at this stage. It's useful to know they're there, though, if only as a reminder of what events are available and as a place from which to start your own investigations.*

Using object (instance) properties

Let's have a quick reminder of movie clip properties in action.

1. Draw a circle on the stage, and with the circle still selected, press F8 to make it a movie clip called `ball`. Give the instance on the stage the instance name `ball01_mc`.
2. Copy the `ball01_mc` instance and paste another instance onto the stage called `ball02_mc`.
3. Create a new layer called actions in the root timeline, and add the following script:

```
ball01_mc.onEnterFrame = function() {
  _x += 1;
};
```

4. Run the movie, and you'll see both balls move across the screen.

Surely you've defined a callback only for the first ball, so why are both moving? It's simple: when you specify `_x` in the event handler, you're actually working with `_root._x`, the x-coordinate of the root timeline, which controls the *whole stage*. Both balls are on `_root`, so both balls move. This occurs because if you don't tell Flash which movie clip you're talking about when you state a property, Flash will search for the nearest timeline, and this is always the one the code is actually attached to. Put in technical terms, Flash will use the **code scope**—in other words, where the code is.

5. Change the code like this:

```
ball01_mc.onEnterFrame = function() {
  ball01_mc._x += 1;
};
```

Now `ball01_mc` moves on its own. This is what you want: the `onEnterFrame` event handler for `ball01_mc` controls `ball01_mc` and nothing else.

6. Of course, that's not to say you can't make it control something else:

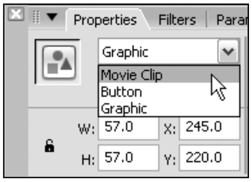
```
ball01_mc.onEnterFrame = function() {
  ball02_mc._x += 1;
};
```

Now the ball01_mc event script controls the _x property of ball02_mc.

It's quite easy to get a feel for a movie clip object, because its properties and methods relate to its physical appearance. As you've seen, not all objects are so intuitive. In general, they can be collections of any number of properties and methods that may or may not have an obvious visual representation.

Symbol types and behaviors

All symbols of a given type such as movie clip or graphic have a certain set of behaviors. A symbol of the movie clip type will behave like a movie clip (with frames, a playhead, and a stop() method), while a graphic symbol has a distinct visual appearance and not much else—certainly nothing that allows you to control it via ActionScript. There's really not a great deal more to know about symbol behavior than common sense tells you. One exception here is when you want to give graphic symbols properties that allow you to animate them, like those the MovieClip class has. To control a graphic symbol from ActionScript, you simply need to use the Property inspector to change its symbol behavior to that of a movie clip by changing the Instance Behavior drop-down from Graphic to Movie Clip:



W: 57.0 X: 245.0
H: 57.0 Y: 220.0

Hang on, though! From the earlier discussion, it sounded like bat_mc was an instance of a class called bat (as defined in the Library). But now you've seen a MovieClip class in the Actions toolbox and discovered it's possible to change the instance behavior without changing the Library item. So what's going on?

Two sides of the same object

The first thing to get clear is this: the Library doesn't *define* object classes. Actually, it just serves as a place where you can collect resources for dropping into your movies. A Library item won't normally have anything to do with the compiled movie unless you put an instance on the stage. If the Library item is set up as a button symbol, it will store resources that you'd expect to find useful in a button, such as four frames labeled Up, Down, Over, and Hit, which will be shown when the appropriate mouse events trigger them. On the other hand, if the Library item is set up as a graphic symbol, it needs only one frame.

So, if you drag a button symbol onto the stage and tell it to behave like a graphic, Flash will create a graphic based on resources from the first frame of the button symbol. Likewise (as you saw just now), if you drag a one-framed graphic symbol onto the stage and tell it to behave like a movie, it won't suddenly acquire more frames—the resources simply aren't there—but it will take on the ability to *behave* like a movie clip. You're basically telling Flash to create a movie clip and use the (limited) resources of the graphic symbol to give it some kind of visual presence on the stage.

This "behaving like a movie clip" business is all part of the `MovieClip` *class. Likewise, a button's behavior is defined by the* `Button` *class. The behavior of a graphic is notable by its absence—it just sits tight and (hopefully) looks good—so you have no need for a class to define what it can do.*

As you can probably see, the Library is a bit of an anomaly when it comes to ActionScript. Symbols in the Library seem to be there for a good reason when you aren't thinking of code (that is, the Library is your asset storehouse), but when you think about classes and instances, the status of symbols in the Library has no real meaning. They aren't yet fully instances, but they aren't classes either. The best way to describe them is as "partly populated instances." They're graphic shells ready to be populated by a class, and this occurs when you move them to the stage.

So, is there any way to tap into these Library resources from your ActionScript? After all, you're trying to find out how to control as many things as possible from ActionScript, and you could save a lot of time and effort if you could make the script drop all your instances onto the stage.

Working with Library items

The good news is that this isn't very hard at all. The main stumbling block is that when Flash generates a SWF, it normally ignores any Library symbols that aren't already being used to help define instances on the stage. This isn't a great problem if you're happy to add an instance of every symbol before you start scripting. Of course, you'd also need to name them all. And hide them all. And be prepared to take a bit of a performance hit if there are lots of them. But, if you have lots of time on your hands and you're not fussy about making your movie efficient, that's just fine. *(Silence.)* Hmm. That's a no, then.

If you've never scripted before (and possibly if you have), this may be the first time you realized this was possible, let alone seen it done. With nonscripted movies, you *always* have to put stuff on the stage manually.

What we're talking about here is quite different: you'll use ActionScript to take a movie clip *directly* from the Library and put it onto the stage. At first this may seem like a minor thing, but you can make some really cool features with this technique. For example, you can let users pick what symbols are used to control a movie—even while it's running. You'll certainly find it lurking behind stacks of the awesome effects produced by famous Flash designers. Let's cut to the chase and see how you can use placing movie clips directly on the stage to create a cool effect.

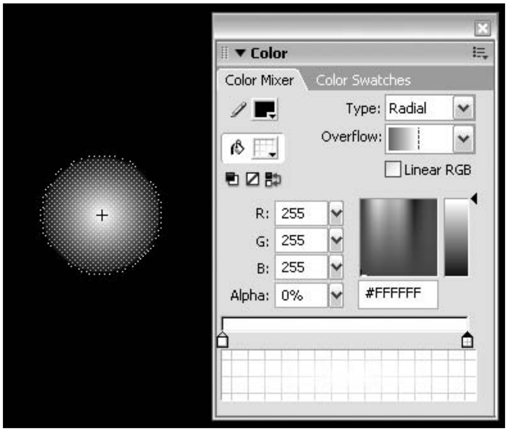
Making a swarm of particles

In this example, you're going to create a small cluster of particles, each of which will move about in tiny, random steps, giving the appearance of a swarming mass of . . . well, who knows what! We hope you'll agree that the end result is sufficiently eye-catching to make the effort worthwhile.

1. Create a new movie (Ctrl+N/Cmd+N) and give it a black background along with a frame rate of 20 fps. In the Timeline panel, rename Layer 1 as actions.

> *The default frame rate of 12 fps is a little dated now, and most computers capable of showing Flash in circulation will handle 20 fps to create smoother graphics.*
>
> *As an aside here, it's interesting to point out that many Flash programmers will push up the frame rate to around or above 60 fps for greater speed and fluidity in games (blisteringly fast* Sonic the Hedgehog, *anyone?). Although this will intensify the gaming experience, you can't be sure that your entire audience will have the processing capability to play the game as it was intended, and it might render the game unplayable on slower machines.*

2. Use Insert ➤ New Symbol to create a new movie clip called `particle`. Use the Color Mixer to set up a radial fill style fading from opaque white at the center to transparent white at the perimeter. Now add a circle to the `particle` movie clip (about 70 pixels in diameter).



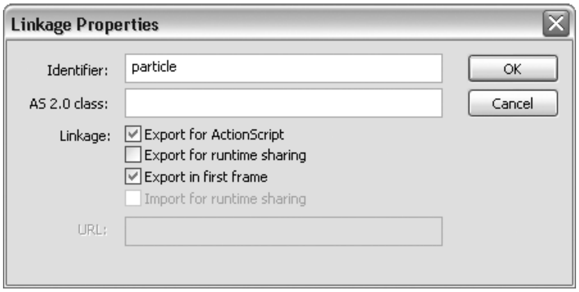
Type: Radial
Overflow:
☐ Linear RGB
R: 255
G: 255
B: 255
Alpha: 0%
#FFFFFF

3. Now click the Scene 1 link to head back to the main timeline.

In a moment, you're going to write some ActionScript that will put several copies of the `particle` movie clip onto the stage. First, though, you must tell Flash to include it in the final SWF and how to identify it. When you create a symbol (graphic, button, or movie clip) in Flash, the name you give it in the Library is really just for your benefit—it's a descriptive name to help remind you what the symbol is for and so you can refer to it easily in text and speech. As far as ActionScript is concerned, that name doesn't exist. So, to call up a Library symbol from ActionScript, you need to give that symbol a special code-centric name that ActionScript can see. This is called a **linkage identifier**.

As you've learned, if there aren't initially any instances of a symbol on the stage (and this will certainly be the case for the particle movie clip), Flash won't normally bother including it in the SWF it generates. If this happened, your ActionScript would end up asking for something that wasn't there. Let's do something about that.

4. Call up the Library panel (press Ctrl+L/Cmd+L) and right-click (Ctrl-click on a Mac) the entry for your particle movie clip. Select Linkage from the context menu that appears, and you'll find yourself presented with the Linkage Properties dialog box. This is where you link your movie clip to the ActionScript environment.
5. Select the Export for ActionScript check box, ensure that the Identifier text box reads particle, and click OK. The movie clip will then get exported to the SWF, ready for ActionScript to play with it. In this case, you've used a linkage identifier that's the same as your descriptive Library name, which saves you from any possibility of a mix-up. Remember, though, that they're quite separate names (one for you, one for ActionScript) and could just as well be completely different if that was what you wanted.



Linkage Properties

Identifier: particle

AS 2.0 class:

Linkage: ☑ Export for ActionScript
☐ Export for runtime sharing
☑ Export in first frame
☐ Import for runtime sharing

URL:

The Export in first frame *check box will automatically be checked when you select the* Export for ActionScript *box. Do not uncheck the* Export in first frame *box!*

6. Select frame 1 of the actions layer (in the root timeline), and with this frame still selected, open the Actions panel. This is where you're going to attach all your ActionScript.
7. Begin by putting a single instance of the particle movie clip onto the stage, using the attachMovie() method:

```
_root.attachMovie("particle", "particle1", 1);
```

This will leave you with an instance of the particle movie clip (called particle1) on level 1 of the root movie timeline. If you test run the movie now, you'll see a portion of the movie clip sitting there, lifeless and still at the top-left corner (because the particle, particle1, has a central registration point, and _x and _y properties of 0, you only see its bottom-right quarter). Let's do something about that and bring it to life.

8. Change the script to the following:

```
var myParticle:MovieClip = this.attachMovie("particle", "particle1", 1);
myParticle.onEnterFrame = function() {
  this._x += (Math.random()*4)-2;
  this._y += (Math.random()*4)-2;
};
```

The first line here creates the particle1 movie clip as before, but it also stores a *reference* to it called myParticle. You can use this reference to attach an onEnterFrame to the just-created clip.

OK, on to the next line. Math.random() generates a random decimal number between 0 and 1, such as 0.56478903454. By using Math.random()*4, you get a random number between 0 and 4, and then subtract 2, giving you a value between –2 and 2.

If you want to be sure you get a whole number (i.e., 1 rather than 1.456734324), you can use Math.round() *to round the value up or down to the nearest whole number. You could also use* Math.ceil() *or* Math.floor() *to simply chop off the decimals and round up or down, respectively. The following code is taken from* math_rounding.fla *from this chapter's download file and displays the usage of said functions:*

```
var val:Number = Math.random()*2;
trace("Original = "+val);
// round up or down
var valRounded:Number = Math.round(val);
trace("Rounded = "+valRounded);
// round up
var valCeiled:Number = Math.ceil(val);
trace("Ceil = "+valCeiled);
// round down
var valFloored:Number = Math.floor(val);
trace("Floor = "+valFloored);
```

Note that you're using a timeline loop to animate the particle. It's only because the movie will, by default, repeat indefinitely that the particles move at all. So what about the ActionScript loop? Well, you actually want to put a whole lot of particles onto the screen. You could just cut and paste the previous code (creating lots of different clips with several attachMovies), but that would be a waste of time when you have the power of loops to fall back on.

9. Add/change the following highlighted lines in your ActionScript:

```
var myParticle:MovieClip;
for (var i = 0; i<10; i++) {
  myParticle = this.attachMovie("particle", "particle", 1);
  myParticle.onEnterFrame = function() {
    this._x += (Math.random()*4)-2;
    this._y += (Math.random()*4)-2;
  };
}
```

Here, you set up a for loop, using the variable i to keep track of how many loops you've been through and using "i is less than 10" as its condition. Everything inside the brackets will be repeated until the expression i<10 turns out to be false. Fortunately, i++ will add 1 to the value of i every cycle, so after ten cycles i will reach 10 and the loop will stop looping.

As you can see, the braces also contain the code you used to create the particle1 instance of your particle movie clip (on level 1 of the root timeline). Now that you're executing that code ten times, it will make ten separate particles, won't it? No, it won't, unless you make a few changes.

First, you need to give each instance a different name, so that ActionScript can tell them apart. Second, you can attach only one instance to each depth level. At the moment, they're all being attached to depth level 1, so each one gets wiped out when the next is attached. You also need to change the depth number every time you call attachMovie().

You already know about one way to do all this, because back in Chapter 5 you did something similar using the function duplicateMovieClip() and a counter variable called i:

```
duplicateMovieClip(particle_mc, "particle"+ i+"_mc", i);
```

In fact, duplicateMovieClip is a MovieClip object method, so you could write this as follows:

```
particle_mc.duplicateMovieClip("particle"+ i+"_mc", i);
```

This time, you don't have an instance on the stage to begin with, so you call attachMovie() from the root movie timeline. Apart from that, though, the same basic principles apply.

So here's how you can attach a brand-new particle movie clip instance to the stage for each new value of i:

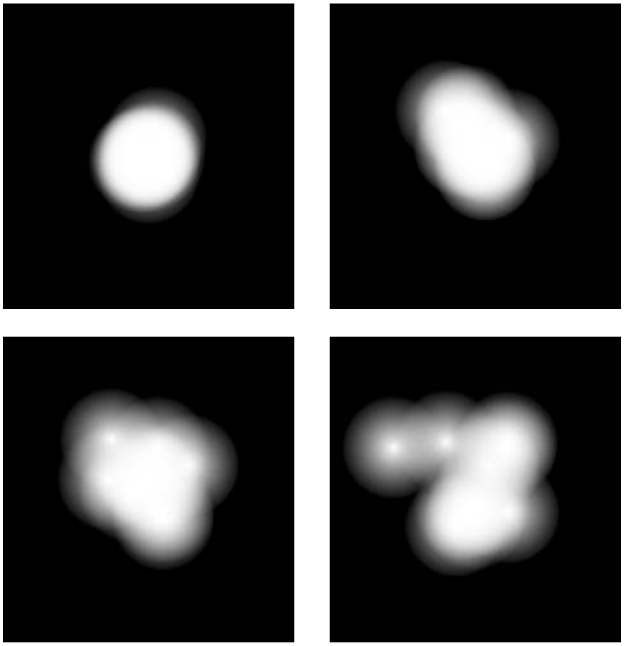
```
myParticle = this.attachMovie("particle", "particle"+i, i);
```

You use + to add (or concatenate) the current value of i to the end of the string literal "particle" to create a new, unique string for each loop.

Finally, add the following line to use the width and height properties of the Stage class to center each particle instance:

```
var myParticle:MovieClip;
for (var i = 0; i<10; i++) {
  myParticle = this.attachMovie("particle", "particle", i);
  myParticle._x = Stage.width / 2;
  myParticle._y = Stage.height / 2;
  myParticle.onEnterFrame = function() {
    this._x += (Math.random()*4)-2;
    this._y += (Math.random()*4)-2;
  };
}
```

You may have already noticed that _root shares many similarities with the movie clips you put into it. In fact, you can almost think of it as a movie clip in its own right: it has a timeline, as well as properties and methods you'd expect from any movie clip. The main difference is that it's always there, so you don't need to create it yourself. If you run the movie now, you'll see the gently pulsating cluster of particles you've been working toward.



Wow—it's a genie!

Let chaos reign

In the eerie effect we've just created we used ten attached instances of our movie clip, each moving independently to create the gentle illusion of life or of something organic. The animation is soft and fluffy, and if you sat and stared at it for long enough, you'd more than likely fall asleep . . . hey, wake up!

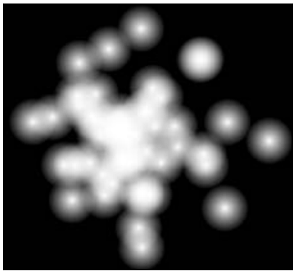
OK, how many of you have already upped the number of duplicates to see what effect that has? If you're anything like us and like to break things, we'd wager that you've already pushed up the figure a little higher than it is meant to go.

Just so that we're all together, change the highlighted lines in the following code to see the change of effect:

```
var myParticle:MovieClip;
var mySpeed:Number = 5;
for (var i = 0; i<50; i++) {
  myParticle = this.attachMovie("particle", "particle" + i, i);
  myParticle._x = Stage.width / 2;
  myParticle._y = Stage.height / 2;
  myParticle.onEnterFrame = function() {
    this._x += (Math.random()*(mySpeed*2))- mySpeed;
    this._y += (Math.random()*(mySpeed*2))- mySpeed;
  };
}
```

In this code we've set the number of duplicates to 50 in the `for` loop, and we've also increased the speed of movement from a potential of 2 pixels to 5 pixels in either direction (+ or -). This speed increase will introduce a little chaos and allow us to see a bit more movement. The actual calculation for the revised random movement is the same as before; we've just changed it so that it is easier to vary the speed with one variable instead of the previous four.

Run the movie to view the changes.

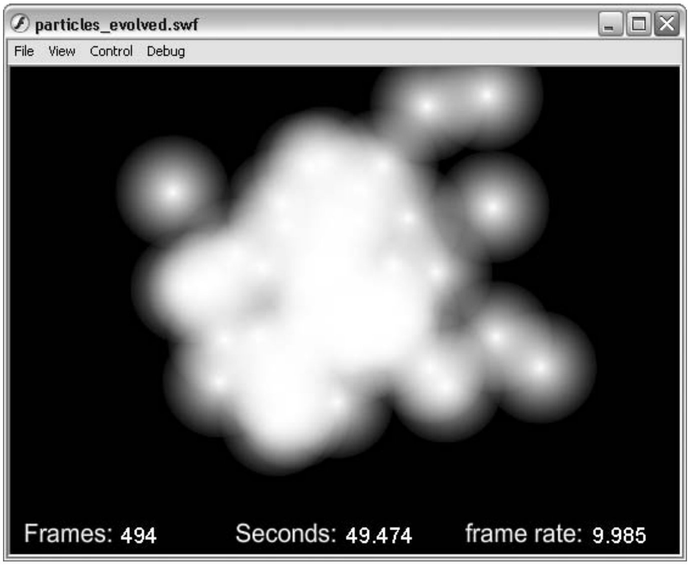


Straight away you'll notice that the individual particles move more erratically, and thus the central "shape" disperses. This freer movement is due to the speed increase, which allows each particle to move further each frame of the movie. Great, isn't it? Well, maybe it's *not* so great. In most cases, unless you're running a machine with a hyper-fast processor, you might have noticed a little slowdown when the movie was running. Although the particles individually move greater distances, the movie has lost its fluidity and smoothness because Flash is collapsing slightly under the pressure and, subsequently, the frame rate has dropped.

If you're one of those fortunate enough to have a brand-new shiny computer and you didn't experience any slowdown, then we recommend you double the number of duplicates to see how the other half lives!

The actual drop in the frame rate of this movie can be approximately calculated with a **frame rate counter**. A frame rate counter will display the active frame rate of the movie and enable you to see any slowdown.

Open and run the Flash movie `particles_evolved.fla`, which contains an instance of a self-contained frame rate counter movie clip on the stage.



Frames: 494 Seconds: 49.474 frame rate: 9.985

Because the movie clip symbol `FPS Counter` *is self-contained, it can be placed on the stage of any Flash movie to provide a reading so you can reuse it in any movie, should you need to.*

Note that the frame rate counter here is displaying an approximate—not exact—frame rate. The frame rate displayed is actually a cumulative figure based on the number of frames *passed divided by the* seconds *elapsed. Generally, a more accurate frame rate counter would take a reading every second, hence* ***frames per second*** *or* ***fps****.*

On Kristian's Pentium 3 laptop, a frame rate of around 8.5 fps (we'll call it "fps" for ease of reference) is achieved after running the movie for 30 seconds. Now if you compare this figure to the actual frame rate of the movie (set to 20, in case you forgot), then it is pretty lacking in performance. Of course, anyone with a Pentium 4 or dual G5 processor should get a significantly higher figure than this, so this is a good reason for having a slow machine in the office, as it allows you to see how your movie will run for those who don't have such speedy machines.

Now, given the speed result on the previously mentioned aching laptop, you'd almost be forgiven if you decided to remove Kristian from your potential target demographic, but luckily you might not have to do this because of a Flash Player 8 enhancement called **bitmap caching**.

Bitmap caching

Perhaps *the* main reason the Flash Player slows down when displaying content is due to the sheer number of objects that it has to draw, redraw, and draw again at very quick intervals. Each frame of the movie, the Flash Player compares the content of the current frame to the previous, and redraws any information that has changed on the screen. If you consider our 50 particles moving every frame, 20 times a second, then the calculations start to hurt.

Here's the science bit. Because vectors are stored as a set of calculations, moving a single shape on the stage means that Flash has to recalculate the vectors each time; this process alone is a complex set of calculations that the Flash Player and our processor has to work out. Ask the Flash Player to calculate the required data for 50 shapes on the screen, and predictably it will struggle to keep up, but it will try!

Bitmap caching helps by taking some of the burden away from the Flash Player to allow it to do other things. It works by converting a specified movie clip's vector information into bitmap data and storing this in the computer's memory. So every time the movie clip is required to be moved on the stage, the bitmap information that is cached is just recalled from memory and displayed. No complex vector calculations and vector drawing are required, and a significant speed increase is the end result.

Enough already, let's quickly see it in action to demonstrate the truly dramatic performance increase.

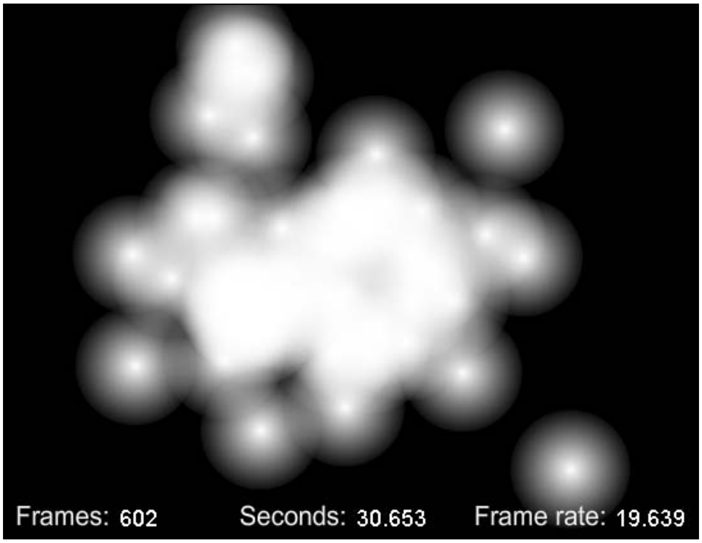
1. In the particles movie that you already have open, add the following line of code where shown:

```
var myParticle:MovieClip;
var mySpeed:Number = 5;
for (var i = 0; i<50; i++) {
  myParticle = this.attachMovie("particle", "particle"+i, i);
  myParticle.cacheAsBitmap = true;
  myParticle._x = Stage.width/2;
  myParticle._y = Stage.height/2;
  myParticle.onEnterFrame = function() {
    this._x += (Math.random()*(mySpeed*2))-mySpeed;
    this._y += (Math.random()*(mySpeed*2))-mySpeed;
  };
}
```

Bitmap caching is a property of the movie clip class, which means that it can be switched on or off at any time during the movie. We've switched it on for every attached instance here.

Bitmap caching can also be specified without ActionScript in the Flash environment using the Use runtime bitmap caching check box in the Property inspector.

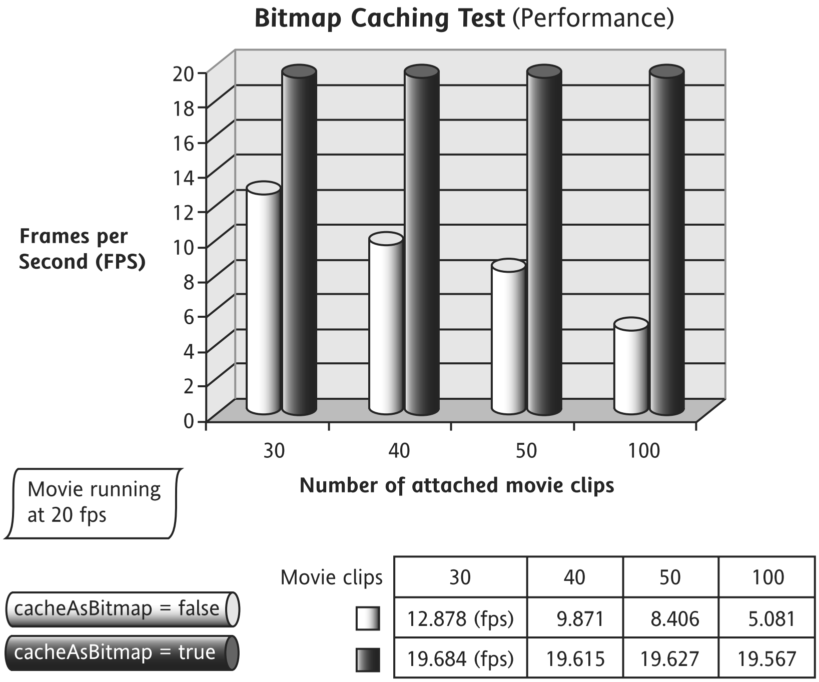
2. Run the movie to see the caching in action.



Frames: 602 Seconds: 30.653 Frame rate: 19.639

Wowee! The speed increase is so significant that Kristian had to check that he was still using the same laptop! The frame rate counter never goes much below 19.5. The reason the performance is so much better is because the Flash Player no longer has to calculate all the vector information for the movie—the required bitmap information is now fetched out from memory. This leaves Flash with a lot less stuff to do, and maybe in time you could teach it to do the ironing or to prepare your lunch, seeing as it has all this spare time to play with.

The following graph illustrates the performance difference between our particles movie with and without bitmap caching. Readings have been taken with 30, 40, 50, and 100 attached movie clips to show how the bitmap-cached version never wavers.



Movie clips	30	40	50	100
□	12.878 (fps)	9.871	8.406	5.081
■	19.684 (fps)	19.615	19.627	19.567

Although the frame rate counter is not as accurate as it could be, the results here are as we would expect. For performance fans: This data was gathered on a Pentium 3 1.2GHz laptop running Windows XP SP2, with the frame rate counter running for 20 seconds in each case.

The other side of bitmap caching

Before you start thinking that bitmap caching makes you bulletproof, there are a small number of simple truths about bitmap caching that we've purposely kept from you while we were selling you the concept. It's important that you know about these before you enable bitmap caching in every movie you create from this day forward:

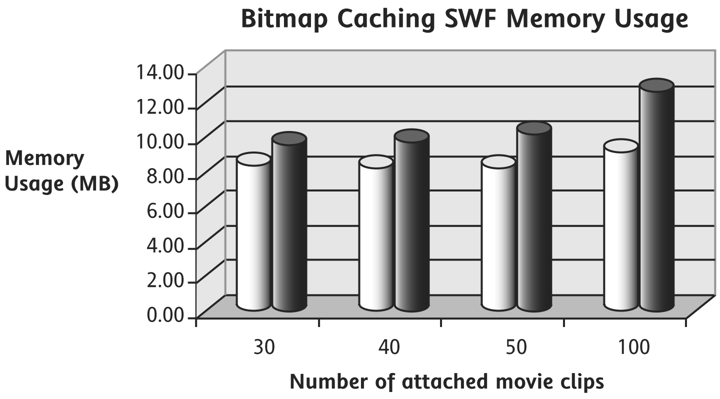
- Bitmap caching can soak up your system's RAM.
- If a bitmap-cached movie clip changes its appearance, then it has to be recached.

Let's look at both factors in detail. Well, there is one other small thing that you might have guessed already: bitmap caching works only in Flash Player 8. If bitmap caching is encountered in older Flash Players, it is just ignored. You also need to make sure that you publish your movies for Flash Player 8 if you are using bitmap caching.

Bitmap caching and RAM

Unless you are reading this book in a hammock in the garden, the Mac or a PC in front of you is only able to run because it has a quota of memory or RAM. A computer's memory is used to store vast amounts of information, from operating system window information, to application states, to Flash variables or arrays. These days, unless you are running a number of RAM-hungry applications at the same time, 256MB or a little more is going to be all you need for most computer tasks. However, if you start working with video or sound, then chances are that you will need a little more memory to cope with the extra data to be stored while you are working.

Generally, Flash Player is not a RAM hog—on the contrary, it is usually very polite. However, with the required RAM storage for use with bitmap caching, you must be aware that Flash Player is obviously going to be a little greedier. In the interest of getting more impressive-looking graphs in the book, we've recorded the amount of RAM taken using our particles movie with and without bitmap caching:



□ cacheAsBitmap = false

■ cacheAsBitmap = true

Movie clips	30	40	50	100
□	8.38 (MB)	8.14	8.29	9.01
■	9.44 (MB)	9.73	10.29	12.61

Each movie was run in the stand-alone Flash Player, and the results were collected after 20 seconds from the Processes *tab of the Windows Task Manager, as shown here:*

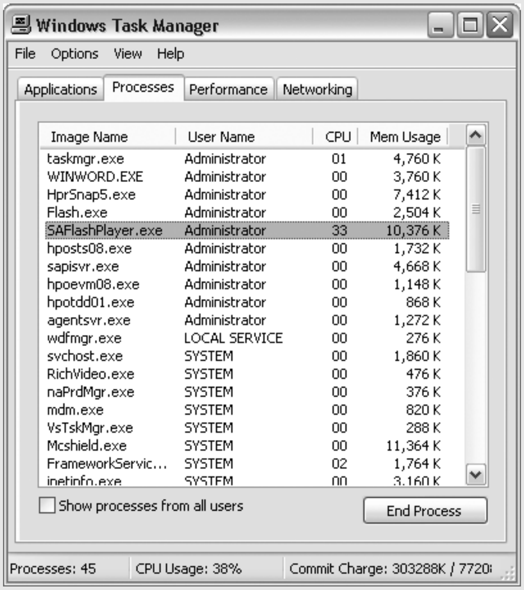


Image Name	User Name	CPU	Mem Usage
taskmgr.exe	Administrator	01	4,760 K
WINWORD.EXE	Administrator	00	3,760 K
HprSnap5.exe	Administrator	00	7,412 K
Flash.exe	Administrator	00	2,504 K
SAFlashPlayer.exe	Administrator	33	10,376 K
hposts08.exe	Administrator	00	1,732 K
sapisvr.exe	Administrator	00	4,668 K
hpoevm08.exe	Administrator	00	1,148 K
hpotdd01.exe	Administrator	00	868 K
agentsvr.exe	Administrator	00	1,272 K
wdfmgr.exe	LOCAL SERVICE	00	276 K
svchost.exe	SYSTEM	00	1,860 K
RichVideo.exe	SYSTEM	00	476 K
naPrdMgr.exe	SYSTEM	00	376 K
mdm.exe	SYSTEM	00	820 K
VsTskMgr.exe	SYSTEM	00	288 K
Mcshield.exe	SYSTEM	00	11,364 K
FrameworkServic...	SYSTEM	02	1,764 K
inetinfo.exe	SYSTEM	00	3,160 K

☐ Show processes from all users

Processes: 45 | CPU Usage: 38% | Commit Charge: 303288K / 7720

As you can see from the graph, caching a number of bitmaps will hog considerably more RAM. As your movie clips get physically larger, the amount of RAM required is likely to balloon because of the way in which the bitmap information is stored: pixel by pixel, without compression or optimization. Given this information, a movie clip with the physical dimensions of 100 × 100 requires the information for 10,000 pixels to be stored in RAM.

However, RAM stealing isn't the only reason that bitmap caching everything on the stage is the wrong thing to do.

Changing the appearance of a cached movie clip

Fortunately, this concept is a little more tangible than the last. As you've already learned, the information from a movie clip is stored in RAM, so that each time Flash Player needs to show the cached movie clip, it will copy the information from memory. Yes? Well, the only drawback to your development is the fact that any changes you make to the appearance of cached movie clips means that they have be recached.

In essence, when you request that Flash Player cache a movie clip, it takes a snapshot of the current appearance of that instance. Flash studies it, converts it into bitmap information, and then sends that data to the computer's memory. Now when you go and change the appearance of that movie clip, Flash Player will have to study it, convert it into a bitmap, and resend the information back to RAM all over again.

The changes that constitute recaching include changes to

- Alpha
- Scale
- Color
- Rotation

Also, the application of filters, the use of the drawing API (this will be covered in Chapter 11), or the change of frame within cached movie clip will also constitute a change of the cached movie clip.

At this point, you are probably thinking, "So what? It's not my problem—let Flash Player deal with it!" Well, if you consider that recaching a movie clip requires a little processing power, then it becomes a problem for the processor to handle all the recaching that you require. Let's see how it affects the performance of our particles movie. (I bet you're sick of those glowing white things by now, aren't you!)

1. Add the following lines shown in bold to the code in your movie:

```
var myParticle:MovieClip;
var mySpeed:Number = 5;
for (var i = 0; i<50; i++) {
  myParticle = this.attachMovie("particle", "particle"+i, i);
  myParticle.cacheAsBitmap = true;
  myParticle._x = Stage.width/2;
  myParticle._y = Stage.height/2;
  myParticle.onEnterFrame = function() {
    this._x += (Math.random()*(mySpeed*2))-mySpeed;
    this._y += (Math.random()*(mySpeed*2))-mySpeed;
    this._xscale += (Math.random()*(mySpeed*2))-mySpeed;
    this._yscale += (Math.random()*(mySpeed*2))-mySpeed;
  };
}
```

2. Run the movie, and you will notice that the frame rate is a bit lower than before. This is because of the added performance hit as Flash Player is being asked to recache up to 50 movie clips every frame.

OK, now that you've learned the two potential disadvantages that can come from overuse of bitmap caching, we'll highlight situations when it is suitable and advisable to use bitmap caching.

When to use bitmap caching

When you look at the frame rates achieved by bitmap-caching our particles, it's fair to say that bitmap caching is a pretty powerful feature that you will no doubt use time and time again to increase the performance of your movies. Games, web applications, and interactive Flash websites will all benefit hugely from the use of bitmap caching, and the added performance opens up a huge number of possibilities for Flash Player.

However, as Flash programmers, we have to be responsible for giving users the best possible experience and performance without any hit on their overall system. With a little careful planning, bitmap caching can be a very good thing for your movies. Here's a list of situations where bitmap caching should be applied:

- Movie clips whose appearance is unlikely to change
- Movie clips with complex vector shapes (as they are often easier to manage when copied from cache rather than drawn by Flash)
- Draggable movie clips
- Movies with a lot of object movement (like our particles movie)

Although the particles movie we've used is a fairly crude example, it does illustrate the performance improvements offered and the potential pitfalls. However, in reality there aren't too many situations where you'll have a movie with 50 movie clips all containing gradients and moving around the stage. But if you do, use the frame counter to see how Flash is performing, and consider using bitmap caching if it will help out.

Now let's say good-bye to our new friend Mr. Caching until the next chapter and resume normal service.

Referencing different timelines with ActionScript

Whenever you use ActionScript to define a variable, a function, or even an object, you're always defining it in the context of a particular timeline. If your script is attached to the root timeline (which it probably will be most of the time), a variable called `myVar` will have the full name `_root.myVar` (or `_level0.myVar` if you look in the debug variables; unless you're into heavy stuff like loading multiple movies, you can assume that `_level0` is always equivalent to `_root`).

The `_root` bit at the beginning is called a **dot path**, and it tells Flash what route it needs to take to get to the correct timeline. As soon as you have several instances of a movie clip in your movie, this becomes very important.

Different place, different variable

Say you define a variable called `igor` in the timeline of a movie clip called `castle`. If you then add an instance of `castle` called `glamis_mc` to the main (`_root`) timeline, the full name of the `igor` variable inside would be `_root.glamis_mc.igor`.

So why go to this trouble? What's wrong with just calling it igor? Well, if you add another instance of castle (called cawdor_mc) to the root timeline, then you'll have another variable called igor sitting in there, too. Its full name would be _root.cawdor_mc.igor, and the two igor variables would be quite separate.

> *Remember, since you address variables in the same way as you address movie clips in a path, it's always handy to include the _mc suffix on the end of the clips' instance names so you can easily tell them apart. Adding _mc to movie clips' instance names will also make them easier to find in your code.*

Most variables are **scoped** to a particular timeline. Identically named variables in different paths are quite distinct entities and can hold different values or types. Flash would recognize your two igors as being different, no problem, but you would have to be very careful to ensure that *you* remember the difference!

Locating variables from inside an event handler function

This seems obvious enough when you're placing your scripts within your movie clips and buttons, but we've already expressed a preference for centralizing code. Ideally, you should keep as much of the code as possible on the first frame of your root timeline. So, how do you locate a variable when you're using it as part of an event handler function?

The normal place to define an event and its handler function is on the root timeline of the movie. Flash will assume that any variables you use inside the function are to be found on that timeline and *not* in the button your callback references. Say you want to model the changing bat population in each of your castles, and you put the following code in the root timeline:

```
glamis_mc.onEnterFrame = function() {
  batCounter = batCounter ++;
};
cawdor_mc.onEnterFrame =function() {
  batCounter = batCounter --;
};
```

From this, you might expect to find Glamis's bat count going through the roof (no pun intended), with Cawdor's quickly plummeting to zero. Actually, both functions are changing the same variable, _root.batCounter (to give it its full name), and because one bumps it up while the other bumps it down every frame, its value never changes. You need to make it explicit that you want each event handler to work on a different variable batCounter, with one defined for each castle, so their full names are glamis_mc.batCounter and cawdor_mc.batCounter:

```
glamis_mc.onEnterFrame = function() {
  glamis_mc.batCounter = glamis_mc.batCounter ++;
};
cawdor_mc.onEnterFrame =function() {
  cawdor_mc.batCounter = cawdor_mc.batCounter --;
};
```

> *A variable has a value and a name, but like streets with identical names in two different cities, a pair of like-named variables can be quite different beasts. If you pick the wrong one, you can get hopelessly confused. For example, say you were in Ealing, London, and you asked a cab driver to take you to Broadway. You might be a little worried if he whisked you off to Manhattan for an audience with Liza Minnelli!*

You now have two variables of the same name but in different places, so they can have different values and not interfere with each other.

Reusing handler functions

Suppose you introduce a third castle into your scenario. This castle is called `rosslyn_mc`, and it has a bat infestation problem similar to that of `glamis_mc`. You might simulate it like this:

```
glamis_mc.onEnterFrame = function() {
  glamis_mc.batCounter = glamis_mc.batCounter ++;
};
cawdor_mc.onEnterFrame = function() {
  cawdor_mc.batCounter = cawdor_mc.batCounter --;
};
rosslyn_mc.onEnterFrame = function() {
  rosslyn_mc.batCounter = rosslyn_mc.batCounter ++;
};
```

Of course, that will work fine, but there's a simpler way to do it. The event handler code you just added is almost exactly the same as what you wrote before for `glamis_mc`; the only difference is that one works on `glamis_mc`, and the other works on `rosslyn_mc`.

The folks in the Flash development team have thought of all this, and they've made it nice and easy to recycle handler functions. There are two steps involved:

1. **Multiple assignments**: Write one event handler to deal with several events.
2. **Relative paths**: Use `this` to refer to the scoped timeline.

> *"So what?" you may ask. "It's just a couple of lines!" Sure, it's no big deal in this case, but if you're writing full-blown movies with event handlers that go on for pages, it's a real pain if you have to duplicate everything and change all the names. As the environmentally conscious among you will appreciate, extra keystrokes make the keys on your keyboard wear out quicker, which means more plastic key production and a swifter onset of global warming. Oh, yeah, plus it makes your movie bigger and generates extra opportunities for code errors—both Very Bad Things as well!*

The first step may sound a bit intimidating, but (as usual) it really isn't! When you write out an event handler, the equal sign does the same thing as it does when you use it to set a variable: it takes whatever's on the right side and assigns it to whatever's on the left side. You may hear programmers refer to the equal sign as the **assignment operator** for precisely this reason ("equal sign" will do fine for us, though). The handy thing is that you can do this more than once. If you write this:

```
myVar1 = myVar2 = "Kris";
```

you're setting the values of two different variables (myVar1 and myVar2) to the string literal "Kris". Likewise, if you write this:

```
glamis_mc.onEnterFrame = rosslyn_mc.onEnterFrame = function() {
  glamis_mc.batCounter = glamis_mc.batCounter ++;
};
```

you're setting the same handler function for the onEnterFrame events of both glamis_mc and rosslyn_mc, and you have to write it out only once.

Hang on, though! If you do this, you're in trouble, since the batCounter variable specified here is the one for glamis_mc. What about rosslyn_mc.batCounter? This is where the second step comes in. You've already used the this keyword. As you may recall, it always refers to the **source** of the event that's being handled—that is, the timeline that generated it. If you're handling the glamis_mc.onEnterFrame event, this points to glamis_mc. Likewise, if you're handling the rosslyn_mc.onEnterFrame event, this points to rosslyn_mc.

So, to make your handler function recycling work properly, you just need to generalize the event handler like this:

```
glamis_mc.onEnterFrame = rosslyn_mc.onEnterFrame = function() {
  this.batCounter = this.batCounter ++;
};
```

Now you have two for the price of one!

Even better, you could make the event handler a general function and then use it on any line you want:

```
function batController ():Void {
  this.batCounter = this.batCounter ++;
};
glamis_mc.onEnterFrame = batController;
rosslyn_mc.onEnterFrame = batController;
```

Notice that you don't refer to the function with batController(), but with batController. This is because you're creating an *association* in this version, rather than asking Flash to *call* the function.

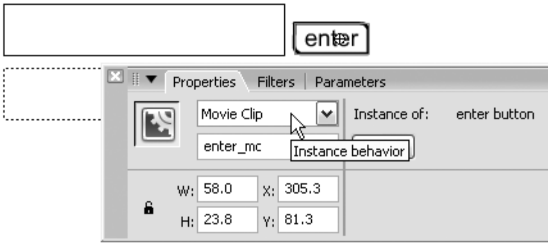
> *Inside an event handler function,* this *in front of a variable name changes its scope from the timeline the code is attached to, to the timeline where the triggering event came from.*

You now have a neat way to keep your code centralized but still have it behave in the logical way that you want it to. Let's see how you can put this to work.

Using the same name, but different variables

Let's start off with our faithful friend, the `inputOutput.fla` movie from Chapter 3 (don't panic—it's included in the download for this chapter). Since you want to look at the effects of nesting variables inside a movie clip, you'll change the enter button to a movie clip.

1. Change the symbol behavior of the enter button instance to Movie Clip via the Property inspector. As a reminder that you're now dealing with a movie clip, give it the instance name `enter_mc`.



Movie Clip | Instance of: enter button | enter_mc | W: 58.0 | X: 305.3 | H: 23.8 | Y: 81.3

2. Select the first frame of the timeline and use the Script pane of the Actions panel to change the script attached to this frame as follows:

```
enter_mc.onRelease = function() {
  input_str = "root";
  output_str = input_str;
};
enter_mc.stop();
```

3. Test the movie and click the button without typing anything. You'll see that both of the boxes display the word root.

> *To write an all-purpose handler, just scope your variables to the controlling timeline (by putting* `this` *in front of their names) rather than to the timeline to which they're attached. This is a fundamental concept you'll need to come to grips with. Once you do, you've taken a major step forward in mastering Flash ActionScript!*

This is because you just told the button's `onRelease` event handler to do that! It sets the input box to display root and then sets the output box to mimic the input box. The extra line you added simply sets the value of `input_str`. Remember that, unless you specify otherwise, a variable exists in the timeline it was created in. At this point, there are no explicit paths in your code, so `input_str` is interpreted as `_root.input_str`, and it's set to hold the string literal `"root"`.

You also have to treat the movie clip slightly differently from the button. A movie clip will start playing as soon as it appears on the timeline, so you have to stop `enter_mc`.

4. Since you define output_str for use by the dynamic text field (which is also sitting in the root timeline), its full name is _root.output_str. You can therefore make this more explicit by rewriting it as follows:

```
enter_mc.onRelease = function() {
  _root.input_str = "root";
  _root.output_str = _root.input_str;
};
```

   Since your script is attached to the root timeline, you don't actually *need* to do this. But what if you want to access variables or properties that *aren't* defined in the root timeline?

5. Add the following lines to the top of your script:

```
enter_mc.onPress = function() {
  this.input_str = "movie";
};
```

   This sets up an event to trigger as soon as the movie clip is clicked, which creates a variable called input_str *inside the movie clip*. It contains the value "movie".

6. Now you can really see what's going on. At the moment, if you test the code you'll still get the root timeline showing in both boxes. Go back to the onRelease handler and change it like this:

```
enter_mc.onRelease = function() {
  _root.input_str = "root";
  _root.output_str = this.input_str;
};
```

   All you've done is added a this path in front of the second variable, but if you test the movie, you'll see what a big difference this makes. The input text box is still being set on the root, so it still displays root, but the output text box is now being set to display the variable you defined inside the movie clip (enter_mc.input_str), so it reads movie.

Now that you've told your enter button to behave as a movie clip, there's no built-in rollover behavior, because that's something a button does, not something a movie clip does.

Well, not unless you tell it to! Let's write some event handlers that will make your movie clip behave like it's a button.

7. First, you can add some actions to your existing handlers. When you click the button, it should go to the down position at frame 2:

```
enter_mc.onPress = function() {
  this.gotoAndStop(2);
  this.input_str = "movie";
};
```

8. Likewise, when you release the button, it should go back to the up state at frame 1:

```
enter_mc.onRelease = function() {
  this.gotoAndStop(1);
  input_str = "root";
  output_str = this.input_str;
};
```

Once again, you've used this to tell Flash what path to use. In both cases, you want to gotoAndStop() on the instance called enter_mc. Since that's the object on which you've defined the handler, you can refer to it using this.

9. You can finish this off by adding rollOver and rollOut event handlers, and making the movie clip stop at frame 1 automatically:

```
enter_mc.onRollOut = function() {
  this.gotoAndStop(1);
};
enter_mc.onRollOver = function() {
  this.gotoAndStop(2);
};
```

Run the movie one more time, and you'll see the movie clip behave just as if it were a button.

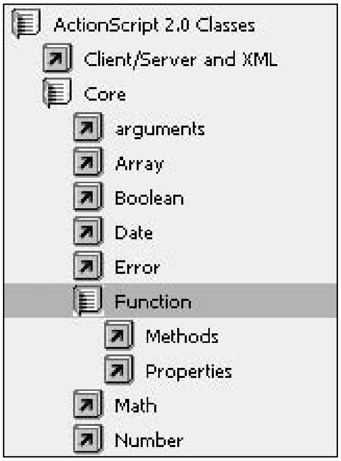
So, a few lines of extra code to get the same effect you had to begin with—what's the big deal? Well, apart from the fact that you've seen how this helps you make sure you're running actions on the right timeline, you now have the framework for a totally customizable button behavior, all written in ActionScript.

Using this script as a starting point, you can make up your own button rules from scratch. What's more, you can have as many complex transitions as you like on this movie clip button—not just the three frames and click zone you have with the normal button.

Let's now move away from buttons and see another use for this.

The apply method

To save typing out the same code a number of times for different objects, you can use a method to add functions to a movie clip (or any object, for that matter). Just as all string variables are automatically able to use the methods defined in the String class, all functions have automatic access to methods in the class called Function. You can find the Function class among the Core objects in the Actions toolbox:



The method you're going to look at here is called apply. This method enables you to take a generic function and apply that to any movie clip you like. Almost like applying our very own created method to a movie clip, huh? Well, although the syntax is different, it's very nearly there. Hold tight, though, because you'll see how to create your own methods in Chapter 15 of this very book.

1. Start a new movie and draw a shape on the stage in the top-left corner. Convert your shape into a movie clip called ball. Give the instance on the stage the instance name ball_mc.
2. Create a new layer called actions above the current layer. Select frame 1 of layer actions and open the Actions panel. Type the following into the Script pane of the Actions panel:

   ```
   function mover () {
     this.onEnterFrame = function() {
       this._x += (_xmouse - this._x)/this._width;
       this._y += (_ymouse - this._y)/this._height;
     };
   }
   ```

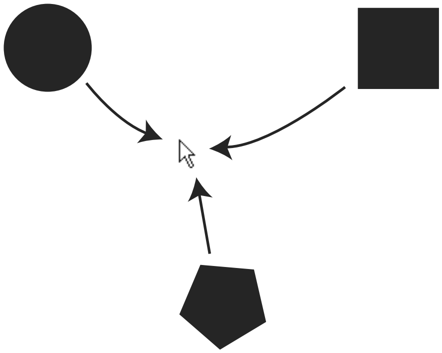
   Here, you embed an event within your main function and just refer to the movie clip as this. One of the most powerful things about the Function.apply method is that it defines this as the object that you're applying the function to, which makes it totally generic. Now to call the function . . .
3. Add this line after the previous piece of code:

   ```
   mover.apply(ball_mc);
   ```

   As you can see, you're addressing your function as you would any other object, using dot notation to specify the method and then passing the argument (the name of the movie clip that you want to apply the function to) inside the parentheses.

That's it! Run the movie and you should see your shape follow the mouse pointer around the screen. For practice, create another couple of shapes on the stage and use the apply method to apply the function to them, too. Depending on the instance names you used, your final code should look something like this:

```
function mover () {
  this.onEnterFrame = function() {
    this._x += (_xmouse - this._x)/this._width;
    this._y += (_ymouse - this._y)/this._height;
  };
}
mover.apply(ball_mc);
mover.apply(square_mc);
mover.apply(polygon_mc);
```



Believe it or not, this was one of the few occasions where we've had the pleasure of using the Polystar tool in Flash. If you have yet to discover said tool, it is accessible below the Rectangle tool in the Tools panel.

Now that you've written the function, you can apply it to any object at all. Use it to move a ball, a tree, a dog—anything you like, really. You can see how powerful this feature is in creating concise and reusable code, and what a benefit it can be to your toolbox. It can initially feel a bit strange thinking of a function as an object, but as with many of these things, the less deeply you think about it, the more sense it makes, and one day you'll be innocently using it when the concept will suddenly click into place and you'll wonder why you ever found it confusing. Flash is often like that, and ActionScript even more so. The secret is to keep at it and continue making the most inane Flash movies, trying strange and different things, until they no longer seem strange and different, just normal.

Global variables

As you've already seen in this chapter, the **scope** of a variable is the part of the movie in which it can be referenced. There are three different types of variable scope, the first two of which are familiar to you:

- **Local**: Variables are available only within the script block braces {} in which they're defined.
- **Timeline**: Variables are available to any timeline (provided you use a target path).
- **Global**: Variables are available to any timeline (and you don't have to use a target path).

You've already seen how you can use `var` to declare **local variables** inside a block of script. You'll most commonly do this when working with loops, which often recycle their counter variables. You don't want `counter` from one loop to interfere with `counter` from another loop, so you make each variable local to its own loop. They're both destroyed as soon as the loop is finished, and neither has any idea of the other's existence.

As for timeline variables, you've already learned that their path is part of what makes them unique; their position in the movie is part of what defines them. Whenever you define a plain old variable, you have to ask *where* you've defined it. Say you define a variable in some ActionScript that you've attached to a movie clip. The variable's **location** will be the timeline of that movie clip. As long as you know where to look path-wise, you can access the variable from anywhere you like.

That's fine, but sometimes it would be nice not to use paths at all. What you really want to do is define a variable that has the entire movie as its scope, so it doesn't need you to specify a target path. Such a variable is called **global**. It has a scope of "everywhere."

Creating a truly global variable in Flash is as simple as this:

```
_global.myVar = "This is available anywhere in the movie!";
```

If you were then to trace this variable from anywhere else in the movie, you could simply write this:

```
trace(myVar);
```

and the correct value would be displayed in the Output window. Try it if you like and see. It's worth noting that you can use the short name only to *access* or *get* the variable, not to *set* it. Say you now add the following code in the root timeline:

```
myVar = "Eric";
```

This would *not* change the global variable myVar to "Eric"; instead, it would create a new variable. Assuming your code is attached to the main timeline, this new variable would be _root.myVar.

If you really wanted to set the global variable to something else later on in the movie, then you'd have to go back to the way you initially did it, by prefacing it with _global:

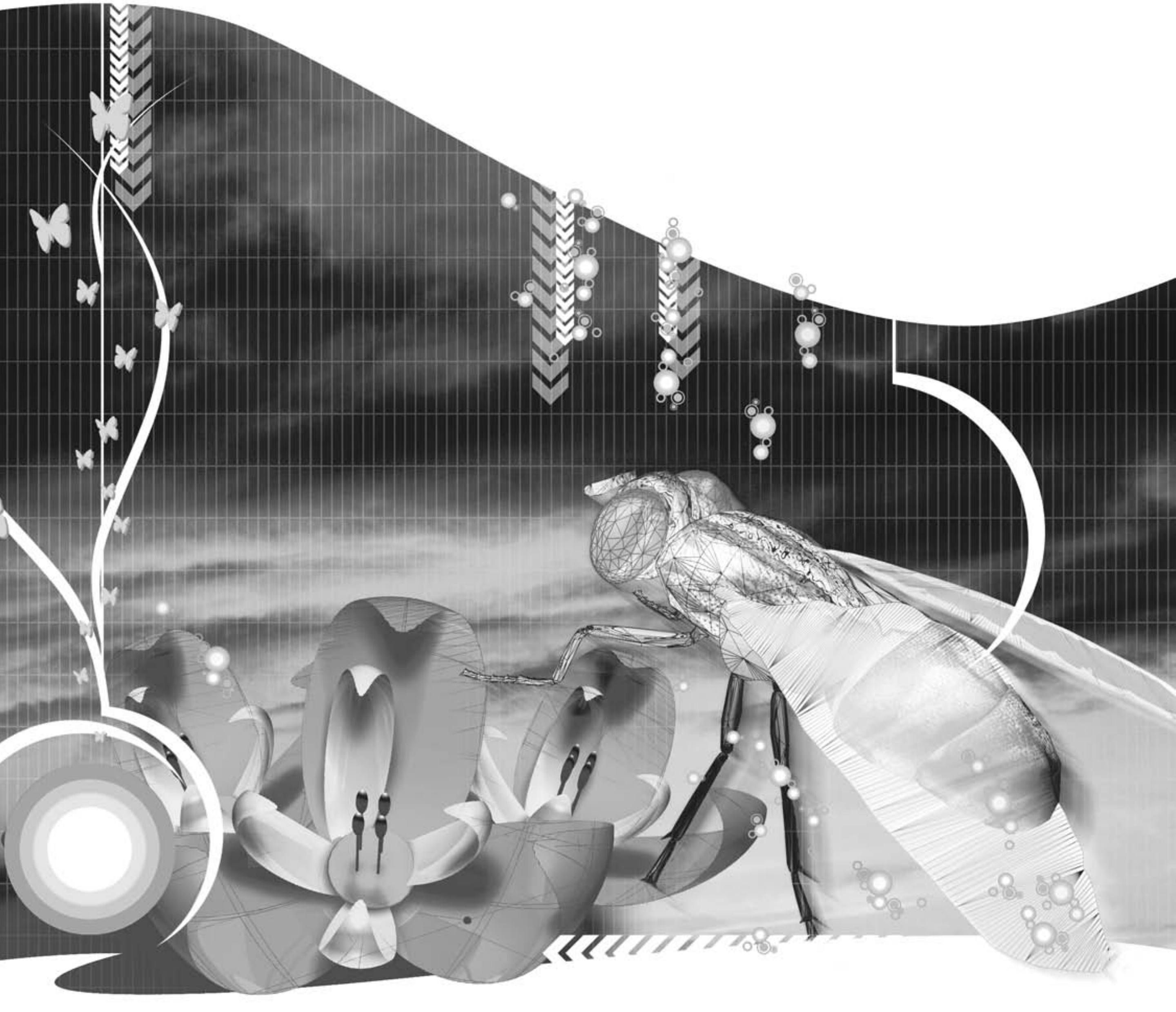
```
_global.myVar = "Eric";
```

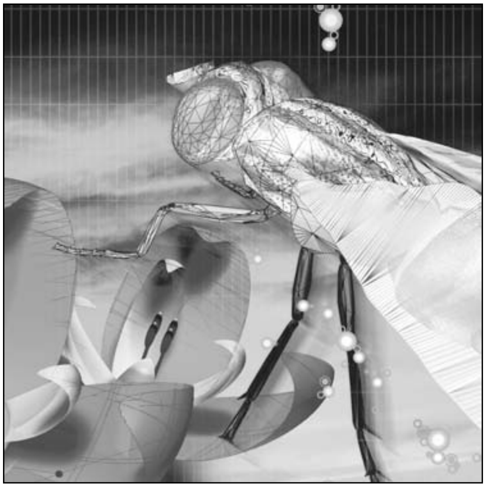
Summary

This chapter covered a wide range of concepts and skills having to do with objects, including the following:

- Using movie clips and buttons as objects, object properties, behaviors, and classes
- Using the Library contents directly through ActionScript
- Using bitmap caching to improve the frame rate of your movies
- Suitable situations for use of bitmap caching and when to avoid overuse of it
- Navigating timelines and using paths
- Using functions as objects
- Creating and using global variables

Congratulations! You're well on your way to mastering the scripting skills that will take your Flash work to new heights. You've learned the basic ActionScript structures that underpin everything else, and you're now ready to explore the amazing stuff that's possible when you put those structures to work. Time for a well-deserved rest! In the next half of the book, you're going to look at building whole projects and adding in cool new features like sound and sprites. The visibility on this mountain is improving with every passing minute, and now it's time to start focusing on the summit.





Chapter 9

REUSABLE CODE AND REALISTIC MOVEMENT

What we'll cover in this chapter:

- Realizing the advantages of creating reusable ActionScript
- Building components that you can reuse in any FLA
- Using a simple formula to simulate realistic movement
- Tweening with ActionScript
- Adding color and movement to the Futuremedia site

No doubt, you want to learn anything and everything to help make your Flash skills commercially valuable, so you can get something back for all the hard work you've put in. You want to perform coding tasks in the minimum amount of time, using the simplest and easiest methods available. What's more, you don't want to get bogged down with irrelevant stuff that you'll never use in the real world.

We wrote this book to help you achieve these aims, and this is where you start getting really serious about those good returns. This chapter holds out the tantalizing prospect of being paid many times over for a single piece of code. Chapter 6 laid the foundations when you learned about modularizing code in functions, but you can take things much further.

Instead of bolting everything together from scratch every time you start work on a new project, you want to keep a bag full of standard bits and pieces that just need a little tweaking to fit right in with what you're doing. This is what modular ActionScript is all about: writing efficient, reusable code. It's efficient, so you need less of it. It's reusable, so you have to write it less often.

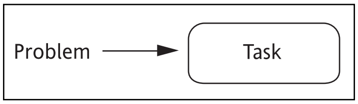
Yes, you've guessed it. We *like* being paid twice for the same piece of coding! You can have lots of identical little movie clips following the same short, simple bit of code, but when taken together, the overall behavior is *complex*. This is a good concept to explore because the code you actually need to write to create this apparent complexity is *short* and *simple*.

This chapter will show you several ways to create reusable blocks of ActionScript that will enable you to make graphic effects you can use over and over again. The examples become progressively more adaptable by making the code increasingly target independent.

The ultimate goal is to build classes of your own, something that you will learn about in the final chapter. Since the introduction of ActionScript 2.0 in 2003, Flash has acquired sophisticated object-oriented programming (OOP) capabilities, but OOP isn't always the most appropriate solution, particularly for relatively small applications. Often a simple solution is all that you need.

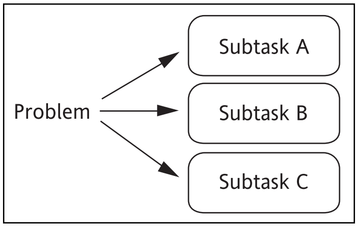
Breaking down big tasks into smaller ones

Many functions and effects require Flash to be doing a number of things at the same time. To do this, you need to work with several timelines, with each one handling a bit of the problem and all of them working at the same time.



Suppose you have a big problem and need to solve it in code. One way is to have an equally big slab of code (the Task block in the diagram on the right) sprawling all over the place in the process. You may end up with an enormous number of lines of code, making it difficult to properly structure, debug, or document your project—and heaven help you when you want to add new functionality!

The next diagram shows the **modular** approach. Here you split the monolithic task into three subtasks: A, B, and C.



This way, you have separate sections of code that handle different parts of your problem. Each subtask is compact, with a well-defined problem to solve, which makes it easy to read and code. Furthermore, since you've broken down the specific task at hand into smaller (and probably more generic) subtasks, you're likely to find these subtasks come in handy elsewhere.

If a similar problem arises in future, you can either reuse your existing subtasks in a slightly different context or tweak them a bit to fit the bill. Either way, you'll wind up spending much less time on development than if you were to write the whole project from the ground up.

Black-box programming

In programming, a **black box** is a piece of code that you can reuse without necessarily knowing all the details of how it works. You can't see inside the box because it's "black" (in other words, its contents are unavailable to you, or **encapsulated** inside the box), but you don't need to. The black-box approach stipulates that all code must be *self-contained* and *distinct* from any other code.

You're interested in only two things here:

- What the code *does* to the outside world (its function or method of working)
- What the code *needs* from you and *gives* back to you (its interfaces)

One example of a black box that should be very familiar is the remote control for your TV. While you may not have the faintest idea of how it works or what goes on inside it, you can still use it to change the channel. You provide it with input (by pressing buttons), it does some processing and talks to the TV, and you get a nice, predictable response: the channel changes.

The black-box concept promotes portability. We're not talking about how easy it is to hold the remote control here, but referring to the ease with which you can carry the remote control's usefulness from one situation to another. If you've used one remote control, you should have a good idea of how all remote controls tend to work. In programming, there are similar advantages that make programming in this way beneficial. If you've cracked one problem with modular code, you've most likely created code that will solve similar problems. You can handle the differences between specific problems by changing one or two modules, but you'll be able to leave a significant part of your code the same.

Additionally, if you're working as part of a team of programmers for a given overall programming project, each programmer on the team doesn't need to know how the others are implementing their allocated parts of the project, so long as the interface specification (that is, the inputs and outputs of the black box) is observed. The TV development team members don't need to know how the remote control unit works; they just need to know about the interfaces (in this case, which signals the remote control will zap at the TV and what the TV needs to do when it senses each one). This concept also lies at the heart of OOP, so by adopting this approach, you are building a solid foundation for the transition to advanced ActionScript.

So how does this black box relate to what you're doing? The first rule is that, to make code self-contained, it really should be within something that separates itself from other code. There are three main ways to do this in Flash:

- **You can use functions within the same overall listing**. You've already done this for other reasons. Creating event handler code and splitting code into separate functions to create modular code entail much the same process.
- **You can "hide" or encapsulate your code in Flash's own black boxes: movie clips**. There are several ways you can do this, but the most modular is via a special kind of black box called a **component**.
- **You can create separate text files containing code only**. Each file is created to perform a specific set of tasks, and you can have a number of such files. You then refer to these files from the FLA at compile time (i.e., when you create the SWF). The code in such external files can be anything (as long as it's valid ActionScript). This is also the way that you create custom classes of your own in ActionScript 2.0.

In this chapter, we'll deal with only the first two ways of creating modular code. You'll see the third method in use in Chapter 15. Another approach, which you'll use in Chapter 13, involves accessing external data—such as an XML file—at runtime, although this is slightly different in that external data affects only what is displayed, rather than affecting the functionality of a movie.

Creating simple components

A component is a Flash-specific black box. It contains code within itself, and sometimes it also talks to other component black boxes to perform its tasks. The beauty of components is that you don't have to look inside them and know how they work to be able to work with them. All you have to know is what the components do and what their interfaces are.

Flash comes complete with a number of ready-built components, which you can see in the Components panel (Window ➤ Components or Ctrl+F7/Cmd+F7). The components that ship with Flash 8 are underpinned with class-based (and rather long) code structures. We'll come back to them in Chapter 15, but let's start the ball rolling with something much simpler instead.

For your component (which is fundamentally just a movie clip) to be useful as a black box, it must be able to control something *other* than itself. In other words, the code inside the movie clip must be able to reach out of the box and control something outside the movie clip. You have to tell your component what it needs to control as well.

There are several ways to do this, but one of the easiest is to make what the movie clip needs to control (the **target**) something general. Rather than name a particular movie clip, you define the target as "whichever movie clip you see at this particular place."

This solution uses a special path called `_parent`, which refers to whichever timeline the movie clip has been attached to. You can think of the hierarchy of movie clips inside a movie as being like a big family tree. The movie itself is the head of the family, which has several children. Each child may (or may not) have children of its own, and so on, and so on. Unlike human families, though, each child can only ever have one parent.

_parent always refers to the movie clip directly one higher in the family tree relative to the current movie clip, so it's a **relative path**. You've already seen one relative path: when this is used inside a function definition, it points to whatever it was that called the function. Now that you're attaching script directly to movie clips, it's useful to know that this can also refer to the movie clip your script is attached to. Just as _parent is like a path to "my mother," this is like a path to "me."

> *It almost goes without saying that you should document black-box code well so you know what interactions to expect when you use it. Good documentation is important at all times, but when you're writing code for later reuse, it's vital. You'll rarely remember the intricacies of something you wrote when you come to reuse it while chasing a deadline 6 months later. Even if it seems like a waste of time now, document it!*

Creating a modular set of playback controls

There are many cartoons and other full-length features on the Web at the moment. If you're creating such features, it's a good idea to design them with the sort of control you get with a normal video. In this section, you'll create a modular set of video controls that you can drop onto any stage containing tween-type animations.

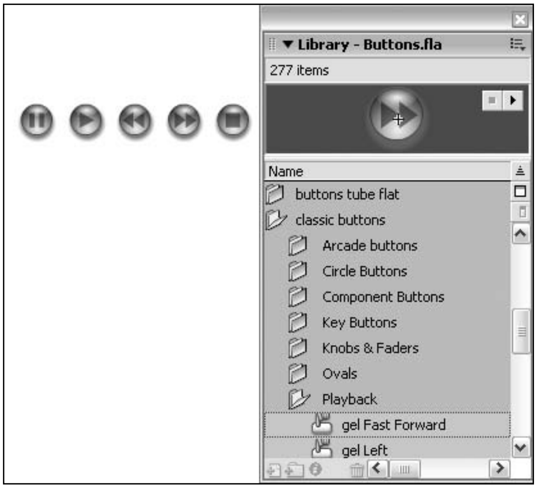
If you get stuck at any point, you can get the finished version, videoControl.fla, from this chapter's download file.

Creating the buttons and black box

Follow these steps to create the buttons and black box.

1. Create a new FLA document. You'll first need the video control buttons. If you want to use the same ones as we did, select Window ➤ Common Libraries ➤ Buttons and then open the classic buttons ➤ Playback folder, where you'll find a set of buttons complete with all the play and rewind symbols.

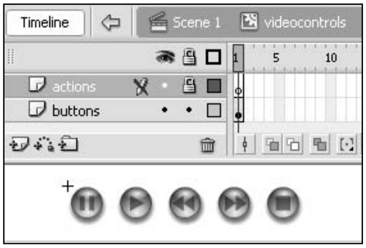
   Drag your buttons from the Library panel onto the stage and arrange them in a strip as shown, giving them the instance names (from left to right) pause_btn, play_btn, rewind_btn, fastForward_btn, and stop_btn.



> *Remember to use fastForward, not fastforward—ActionScript is case sensitive!*

This strip consists of five buttons. The Pause button should freeze the target timeline at the current frame. Then, every time it's clicked, it should advance to the next frame. The Play button should undo any other button and start to play the movie from the current frame. The Rewind and Fast Forward buttons should work like standard video fast-search buttons and thus cause the movie to play at a faster rate (say ×4), either backward or forward. The Stop button should stop the movie at its current frame.

2. Select all of the buttons (either drag a selection box around them or hold down the Shift key and click each in turn until you've selected them all). Press F8 to convert them to a movie clip called `videocontrols`. Now double-click the movie clip to enter edit-in-place mode. Rename the layer containing your buttons as buttons, and create a brand-new layer called actions. As always, remember to lock the actions layer.



Let's start defining actions for these buttons. The Play and Stop buttons should be relatively painless, so you'll look at them first.

Adding the play and stop code

In this exercise, you'll add the play and stop code.

1. The Play button simply has to make the timeline on which your video controls sit (which will be their `_parent` timeline) start to play. All you need to make this happen is an onRelease event handler containing a `_parent.play()` command.

   Pin the Actions panel to frame 1 of layer actions and add the following script. You'll add all further scripts as a continuation of this script, so there's no need to unpin for a while yet.

```
play_btn.onRelease = function() {
  _parent.play();
};
```

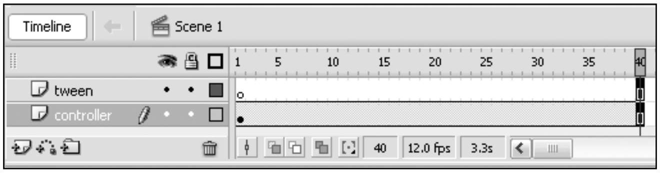
2. The Stop button isn't much different from the Play button, except that it needs a `_parent.stop()`:

```
stop_btn.onRelease = function() {
  _parent.stop();
};
```

Testing the SWF so far

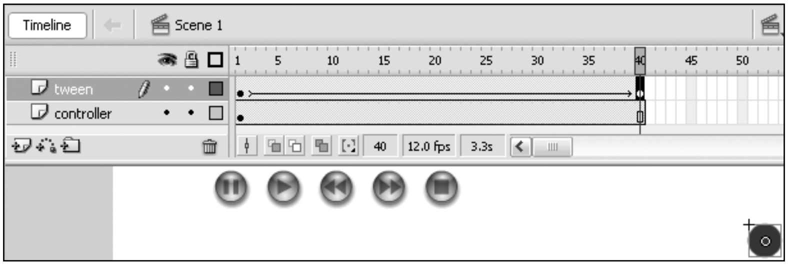
Now that you have a little functionality to play with, you'll try out the controls. You're going to add a simple motion tween so that you can see when the timeline is running and when it isn't. A little later in the chapter, you'll use it with a complex movie that's far more interesting than a simple tween. However, working with the tween first makes it easier to grasp the concept of what's going on.

1. Go back to the main timeline, rename the existing layer as controller, and add a new layer called tween. Select frame 40 in both layers and press F5 to insert frames.



2. Select frame 1 on the tween layer and draw a circle on the left of the stage just below the row of buttons. With the circle still selected, press F8 and make the circle a graphic symbol called `circle`. Select frame 40 in the same layer, choose Insert ➤ Timeline ➤ Create Motion Tween, and press F6 to insert a keyframe.

   With this frame still selected, Shift-drag the circle over to the right of the stage.



3. Run the movie to see the tween in action, and try clicking the Stop and Play buttons to check that they work.

Now you'll move on to some slightly trickier code.

Adding the Pause button code

The Pause button needs to do one of two things. If the movie is playing, clicking the Pause button should bring the movie to a halt, just as clicking the Stop button does. However, if the Pause button is clicked when the movie is *already* paused, it should do what many video recorders do, something called **single stepping**. Every time you click the Pause button after the initial pause, the movie moves forward by a single frame and then stops again.

Let's say you're sneaky about this, and you enter the single step mode right away (i.e., you go to the next frame and pause for *all* Pause button clicks). How many people will actually notice? How many people have ever noticed that video recorders do two different things on the first and second press of the Pause button? That's right: none, because the frame rates are too fast for anybody to realize what's happening.

The onPress action for the Pause button is actually the same as that for the Stop button: a simple _parent.stop(). For the onRelease action you want the _parent timeline to go to its next frame and stop. Conveniently, there is an action to do this: nextFrame(). The code _parent.nextFrame() will send the timeline that videocontrols is on to the next frame.

Add the following code to the listing:

```
pause_btn.onPress = function() {
  _parent.stop();
};
pause_btn.onRelease = function() {
  _parent.nextFrame();
};
```

Adding the fast forward and rewind code

The Fast Forward and Rewind buttons are also a little tricky because you want them to change the speed of the movie for as long as they're held down. The trouble is, Flash doesn't give you any controls to change the frames per second. If you want to play the movie at three times the normal speed, you must do this by playing just one frame out of every three.

You can achieve this by writing event handlers for the onEnterFrame event in the videocontrols timeline. Every time the playhead enters a frame, it should kick the parent's timeline on to the next frame but two (if you're fast forwarding) or the last frame but two (if you're rewinding). These behaviors should kick in when the relevant button is clicked, and they should switch off as soon as that button is released.

This gives you three possible tasks for the onEnterFrame event handler:

- **Normal mode**: Do nothing.
- **Fast forward mode**: Go to the current frame plus 3.
- **Rewind mode**: Go to the current frame minus 3.

As you know, you can't define multiple event handlers for a single event, so how do you make onEnterFrame behave properly in each circumstance? Well, you could define a variable to specify which mode you're in, use your button events to set it, and write an all-purpose handler (perhaps using a switch...case structure) that picks appropriate actions based on its value.

Then again, why not cut out the intermediary? You want to use button events to modify what the handler does, and there's no overlap between the different modes and what they do. Let's just get your button event handlers to completely *redefine* the onEnterFrame handler as required.

There's an important distinction to note here: onPress and onRelease are both user-generated, "one-shot" events, whereas onEnterFrame is generated by the timeline and fired for every frame. You can think of onPress and onRelease as manual, whereas onEnterFrame is automatic (or internally generated by Flash events). By using one to control the other, you're producing some pretty complex behavior. When the user causes a button event (such as onPress and onRelease), Flash will run the associated button event handlers, but these handlers can themselves set up *other* events such as onEnterFrame.

This relationship between user-generated and Flash-generated events is one of the core tricks in advanced motion-graphics programming. Clicking a button doesn't necessarily just have to cause navigation by jumping around the timeline. It can also cause new events to become active, and these new events cause the real responses to the initial button event to take place. The reason for doing this is that the initiating event (button event) may not be the best event handler for your response to the button click. The Futuremedia site contains a good example of this, where clicking a colored strip causes several things to happen in sequence. You'll see the code associated with this start to take shape at the end of this chapter. The simple video controls you're developing here are a precursor to it.

1. You'll define each of the `onPress` button event handlers so that they set up a handler for `onEnterFrame` that will fast forward or rewind the parent timeline. The buttons' `onRelease` handlers then delete the timeline's `onEnterFrame` handler. Add the following code in the Actions panel:

```
fastForward_btn.onPress = function() {
  onEnterFrame = function() {
    // if fewer than three frames left, go to last frame
    if (_parent._currentframe+3 > _parent._totalframes) {
      _parent.gotoAndStop(_parent._totalframes);
    } else {
      _parent.gotoAndStop(_parent._currentframe+3);
    }
  };
};
```

Remember: `fastForward`, *not* `fastforward`*!*

This script defines an `onEnterFrame` handler for the current timeline within the button event handler, making the parent timeline advance at three times the normal speed. This will continue indefinitely until you redefine or remove the `onEnterFrame` handler.

The script works by examining at which point the parent timeline is by looking at the property `_currentframe`. This property contains the current frame number. As well as reading it, you can also *change* it, and that's what the code does to give you the "skip frames so that the animation runs faster" effect.

We mentioned earlier that ×4 would be a good approximation of a faster rate for a fast forward or rewind function. When you're calculating this, you need to bear in mind that Flash advances one frame in the time it takes for the script to run anyway, so for a Fast Forward button, you want Flash to advance by three frames to make up the four frames for a fast forward.

As you may remember from the discussion of flowcharts in Chapter 2, it's a good idea to check whether adding 3 to the value of _currentframe will exceed the number of the final frame (which can be accessed through the _totalframes property). If this is the case, the first half of the if statement sends the playhead to the final frame. Otherwise it goes to _currentframe+3.

Notice that the code specifies the _parent path for both the gotoAndStop() method *and* the _totalframes and _currentframe properties.

> *As an aside, the* onEnterFrame *handler is attached to the timeline of* videocontrols*, not the* _parent *timeline. You do this because in the black-box approach you keep all your code in the black box (the* videocontrols *movie clip) and try not to affect the* _parent *timeline via anything other than the video-controlling features.*

2. To stop fast forwarding, you need to delete the onEnterFrame script. This happens the instant the Fast Forward button is released.

```
fastForward_btn.onRelease = function() {
  delete onEnterFrame;
};
```

3. Now that both event handlers have been created for the Fast Forward button, test the movie again to check your progress.

4. The Rewind button is very similar, but needs to check whether it's likely to overshoot the beginning of the timeline (which Flash Player won't let it do). This is what it looks like:

```
rewind_btn.onPress = function() {
  onEnterFrame = function() {
    if (_parent._currentframe-5 <= 1) {
      // if fewer than five frames left, go to first frame
      _parent.gotoAndStop(1);
    } else {
      _parent.gotoAndStop(_parent._currentFrame-5);
    }
  };
};
rewind_btn.onRelease = function() {
  delete onEnterFrame;
};
```

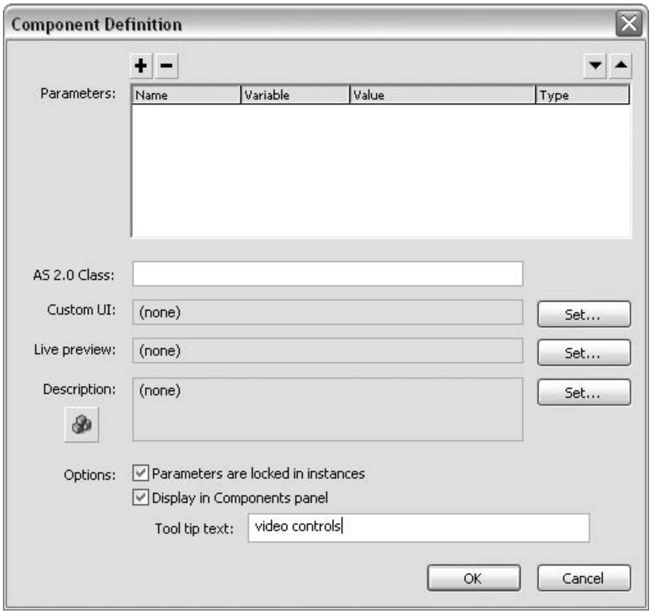
This does exactly the same for the Rewind button as the Fast Forward button, except that you move backward. You want to move backward by four frames at a time, but because the timeline may actually be moving forward one frame per second, you subtract 5. You also need to check whether the playhead can actually move that far back, because gotoAndStop() won't accept any number less than 1. If _parent._currentframe-5 is 1 or less, you need to send the playhead to the parent's first frame.

Now that you've defined actions for all your buttons, you can use the videocontrols movie clip to control any timeline it's thrown at. After you have tested it with the tween, we'll show you how you can place it on *any* movie clip, and it will work in the same way. Here's the full code listing before you do this.

```
play_btn.onRelease = function() {
  _parent.play();
};
stop_btn.onRelease = function() {
  _parent.stop();
};
pause_btn.onPress = function() {
  _parent.stop();
};
pause_btn.onRelease = function() {
  _parent.nextFrame();
};
fastForward_btn.onPress = function() {
  onEnterFrame = function() {
    // if fewer than three frames left, go to last frame
    if (_parent._currentframe+3 > _parent._totalframes) {
      _parent.gotoAndStop(_parent._totalframes);
    } else {
      _parent.gotoAndStop(_parent._currentFrame+3);
    }
  };
};
fastForward_btn.onRelease = function() {
  delete onEnterFrame;
};
rewind_btn.onPress = function() {
  onEnterFrame = function() {
    if (_parent._currentframe-5 <= 1) {
      // if fewer than five frames left, go to first frame
      _parent.gotoAndStop(1);
    } else {
      _parent.gotoAndStop(_parent._currentFrame-5);
    }
  };
};
rewind_btn.onRelease = function() {
  delete onEnterFrame;
};
```

5. There's one more thing you need to do: make your movie clip into a component. Open the Library panel (Ctrl+L/Cmd+L or Window ➤ Library), click the New Folder icon to add a new folder, and name it videocontrols assets. Drag all the gel buttons into it. Now right-click (or Ctrl-click) the videocontrols movie clip in the Library and select Component Definition. This will bring up the Component Definition dialog box.

6. Tick the Display in Components panel check box and give it the tooltip text video controls, as shown in the following screenshot.



Component Definition

Parameters:

Name	Variable	Value	Type

AS 2.0 Class:

Custom UI: (none)

Live preview: (none)

Description: (none)

Options: ☑ Parameters are locked in instances

☑ Display in Components panel

Tool tip text: video controls

Click OK, and the movie clip should now appear in the Library as a component.

7. Save your file as `videoControl.fla` and leave it open for the next exercise.

You can now use this set of controls on *any* timeline that consists of tween or timeline animations, anywhere, in any FLA. All you have to do is drop it into the target timeline! So how have you achieved this feat of coding?

Well, your video controls don't refer to any *specific* timeline, but to `_parent` (i.e., whatever timeline `videocontrols` finds itself placed on). The controls don't need to know anything about this timeline—not even how many frames it has. Since the controls are completely self-contained, you can drag them onto *any* timeline and they'll work.

> *As well as being self-contained (or inside a black box), well-written modular code has to be general. In this exercise, you achieve this by using the* `_parent` *path to define the target timeline.*

The fact that you've wrapped the whole thing up as a component means that you can drag `videocontrols` from its place in the Library and onto any other timeline, whether it's in the same movie or not. All its assets (i.e., the bits and pieces it needs; in this case, the gel buttons) are automatically copied across with it.

You now have a completely self-contained building block, so you should be able to reuse it elsewhere. Let's try this out.

> *Although the* videocontrols *movie clip is displayed as a component in the* Library *panel of the current FLA, it won't automatically appear in the* Flash 8 Components *panel. For that to happen, you need to export a SWC file and copy it to your Flash* Configuration *folder. You can find full instructions at* http://livedocs.macromedia.com/flash/8/main/00003056.html.

Dark Valentine

A few years ago, Sham started a little side project of creating a deck of playing cards based around photography and Photoshop manipulation. He collaborated with a photographer named Martinus Versfeld for one of the images called "Dark Valentine." This was his initial set of interpretations for the queen of hearts:

beauty is only skin deep
BEAUTY IS ONLY SKIN DEEP

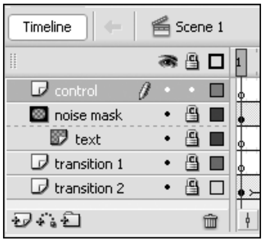
BUT WITH HER. WITH HER...
IT STARTS FROM THE BONE

At around the same time, people were beginning to use Flash for e-greeting cards. Most of the cards were pretty saccharine, and Sham thought images like Dark Valentine would make a nice antidote, so he set about creating a simple Flash animation using the Photoshop images as a base. You can find the results in the download for this chapter as `dark.fla`.

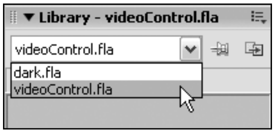
> *Because the Dark Valentine work uses some rather hard-to-find fonts, they have been converted to vectors so all readers will see the same animation irrespective of whether they have the fonts installed. This has the side effect of increasing the file size of the animation, but the actual version (with the fonts correctly embedded and "real text" used throughout) was designed to stream correctly on a 56K connection.*

Using your component with Dark Valentine

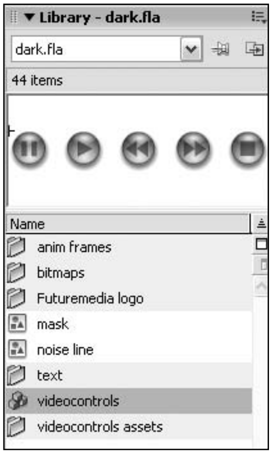
This exercise will show you how easy it is to drop video controls onto another timeline, by using them to control the Dark Valentine animation.



1. Open `dark.fla` and create a new layer called control on the main timeline. This is where you're going to put your controls. Make sure that it's the topmost layer because you want your video controls to appear over the Dark Valentine animation. All the other layers are locked, although you're welcome to investigate the contents later to see how it's all put together.



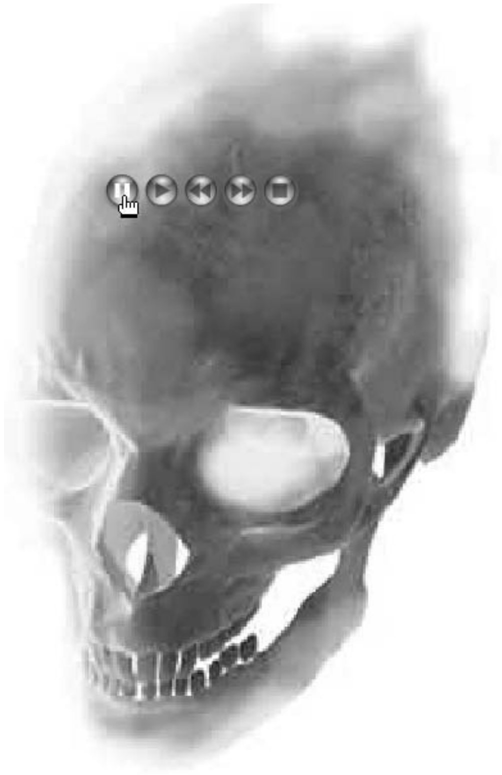
2. Select the control layer. Then activate the drop-down menu in the Library panel (as shown in the screenshot) to select `videoControl.fla`, which should still be open from the previous exercise. This opens the `videoControl.fla` Library, while leaving `dark.fla` selected in the workspace—a new feature in Flash 8 that makes it very easy to import movie clips and other assets from another FLA.



3. Drag the `videocontrols` component onto the stage and locate it in a suitable position to control the movie.
4. Use the drop-down menu in the Library panel to switch back to the `dark.fla` Library. You'll now see the `videocontrols` component listed along with the `videocontrols assets` folder (which contains definitions for all the gel buttons).

   Yes, a tidy Library is a happy Library. It's always a good idea to keep things neat in a document as complicated as this, and you don't need us to tell you why!

5. If you run the dark.fla movie now, the video controls will operate as expected. You can even use the controls in pause mode to see that scary bit where the skull X-ray appears suddenly.



Eek!

Of course, this set of controls will act only on a single timeline. If your animation consists of other movie clips as well as the timeline that holds videocontrols, it won't control them, just the timeline that it's placed on. You could modify the controls so that they can be hidden. You could even add a drag bar or use the _alpha property to make the controls semitransparent when they're not in use. You could build up a whole site navigation system using a little TV remote control unit for your navigation—click the channel buttons to move about, and use the video controls to control the movies. Enough of our ideas, though. We'll leave you to experiment.

You've now seen how to build a reusable component, which gives you a convenient timeline controller that you can use in all sorts of different situations. This has lots of consequences for how you write your Flash movies in the future, but so far you've seen only the tip of the iceberg.

Modular control of movie clips

Imagine you have a movie clip called `blob` and an instance of it called `blob_mc` on the stage. Say you want to make it do several things at the same time. Perhaps you want it to move around the screen in a particular way (based on one set of criteria), but you also want to change its appearance (based on a completely separate set of criteria) as it moves. The simplest solution would be to write a couple of functions that refer specifically to `blob_mc` and trigger them whenever the relevant criteria are met. The better way is to write a set of general functions that can refer to *any* movie clip.

Of course, the other "other way" Flash implements modular code in a drag-and-drop fashion is via behaviors. The behaviors that ship with Flash 8 are aimed at raw beginners and aren't designed for people with any coding experience. For this reason, they're not covered at all in this book.

You'll look at arranging code into modules using inheritance and code-based hierarchies (as well as timeline-based parent/child hierarchies) when you continue the Futuremedia book project later in this chapter. And in Chapter 15, you'll explore using high-end code to achieve modularity with ActionScript 2.0 classes. You certainly won't be a beginner any more when you get to that point! But for now, let's limber up by sticking to a more graphical, component/movie clip–based way of looking at modularity and general code.

Supposing you have a number of blobs, all of which you want to move in a slightly different way. You *could* use a different component per blob, but a better way is to use components that allow configuration via *arguments*. You'll develop this feature in the following section.

How to simulate realistic movement

Before you look at coding, you're going to take a quick detour. You're well on your way to the top of the ActionScript mountain, so you won't settle for just *any* functionality here—you'll go for the big one: **real movement**. This is motion based on real physics, and it takes into account the fact that objects in the real world have mass and therefore move in a fairly complicated way. Let's work out what number-crunching code needs to go into your component before you start building it.

Don't panic, though! There's very little math involved here; there's just a lot of common sense, along with some precise observations (and, because Flash isn't a graphics speed-demon, a lot of approximation). If you find it hard going, just take it slowly. It's not difficult—just a bit different—and it's well worth knowing about if you want to go on to build the kind of sites and effects that have everyone gaping in awe.

To learn a lot more about creating realistic movement with ActionScript, grab hold of a copy of Foundation ActionScript Animation: Making Things Move! *by Keith Peters (friends of ED, ISBN 1-59059-518-1).*

Creating realistic motion isn't that different from what you've already done in Flash, but it can seem quite challenging, simply because you have to account for little things like the laws of physics. OK, come out from behind the couch. It may sound scary, but it's not. Really.

Real objects have **mass**. This is what makes them feel heavy, and it's also what makes it hard to start and stop them moving. Just think about how much less effective your accelerator and brake pedals are when your car is full of people. For a more dramatic example, just think about pushing a massive oil tanker along a flat stretch of road with your bare hands—or worse still, trying to stop it!

Objects have mass, so they can't start (or stop) moving immediately. When they start to move, it takes time for them to build up speed. When they stop moving, it takes time for them to come to a halt. This is called **inertia**.

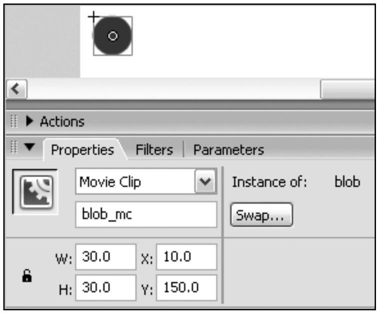
You'll be pleased to know that this is all the physics you really need for this simulation. Of course, if you let it, this could start getting all mathematical and tricky. You don't really want that, so let's break down the process into some commonsense steps. Common sense is well known as an ancient and powerful approach to problem solving, and it's usually less scary than physics.

Let's create the blob movie clip and slowly develop some motion code as you go along.

Animating linear motion

Say you place an instance of the blob movie clip (called blob_mc) on the stage at x-coordinate 10. You want to move it across to x-coordinate 200 in a smooth movement.

1. Open a new Flash movie and select Modify ➤ Document (alternatively, just double-click the area at the bottom of the timeline, where the frame rate is listed). In the Document Properties window that appears, set the frame rate to 20 fps. This is a normal frame rate for smooth animation.
2. Draw a small (around 30×30 pixels in size) filled circle, then select it and press F8, making the circle into a movie clip called blob. Give the version on the stage an instance name of blob_mc, and use the Property inspector to give it an X position of 10. The Y position isn't important, but 150 is as good a place as any. This should place your blob at the left edge of the stage.



Movie Clip
Instance of: blob
blob_mc
W: 30.0
X: 10.0
H: 30.0
Y: 150.0

3. Rename the current layer as graphics, and add another layer above it named actions.
4. Select frame 1 of layer actions, and enter the following code in the Actions panel:

```
blob_mc.onEnterFrame = function() {
  if (this._x < 200) {
    this._x += 4;
  }
};
```

Every time you enter a new frame, the event-handler code checks whether blob_mc's x-coordinate is less than 200, and it adds 4 if it is. When the movie starts playing, the x-coordinate is 10 (where you originally put the blob). After one frame blob_mc is at x-coordinate 14, after two it's at 18, and so on. Eventually its x-coordinate exceeds 200 and it stops moving (at x-coordinate 202 to be precise).

Well, that was easy! If you try it out, though (and you can if you run blob1.fla from the download), you'll see that it moves quite unnaturally—it moves at the same speed throughout and stops abruptly. This makes it look a little fake.

Motion with acceleration

So how can you fix this? Let's pause for a moment and think about what you're trying to do. You'll keep it simple for now: make the blob slow down before it reaches the target. At the moment, it moves at the same speed throughout. The line that does this is

```
this._x += 4;
```

which moves the blob by 4 pixels for every frame. What if you change 4 to something else? Well, changing it to 1 will slow the whole thing down, and changing it to 20 will speed the whole thing up, but neither would actually do what you want and give the blob different speeds at different times. What you want is something that can vary—a variable!

Let's say you use a variable called speed to tell you how fast to change the position:

```
this._x += speed;
```

Now you just need to figure out what value to give speed. You know where to start looking: you want the blob to begin at X:10 with a speed of 0 and end up at X:200 with a speed of 0. Of course, you can also guess that it needs some more speed in between; otherwise, it won't ever get to X:200!

You need to find a value that starts positive (when the movie starts) and then drops away to 0 as the blob approaches its destination. That's easy: you use the distance between the blob and the target. You can work this out by finding the difference between blob_mc._x (the blob's current x-coordinate) and target (the x-coordinate of the destination). When you do the sums, you find it starts off at 190 (200 – 10 = 190) and finishes as 0 (200 – 200 = 0), which is just what you need.

You know where the blob is heading, so the last bit should be easy. You have the following equations and values:

```
target = 200
speed = target - _x
this._x += speed
```

So, if _x is less than 200, the speed is positive and the blob moves to the right. If _x is 200, the speed is 0. Just what you wanted, isn't it? Not quite. You see, the first time your movie works out the value of speed, it will be 190. Add that to _x, and it will take you straight to the end, before you have time to blink. If you're in any doubt, try it yourself by rewriting the code as follows and testing the movie:

```
blob_mc.onEnterFrame = function() {
  if (this._x < target) {
    speed = target - this._x;
    this._x += speed;
  }
};
var speed:Number = 0;
var target:Number = 200;
```

OK, then, let's scale down the speed a little by making it one-tenth of the distance. Change the line that sets speed as shown:

```
blob_mc.onEnterFrame = function() {
  if (this._x < target) {
    speed = (target - this._x)/10;
    this._x += speed;
  }
};
var speed:Number = 0;
var target:Number = 200;
```

Now the speed value will start out as 10 and get smaller as the blob closes in on its target. This approach gives great results—check out blob2.fla to see this simple example in action (if you aren't following along by creating your own FLAs).

Let's consider what you've done here. As the blob gets closer to its destination, you want it to slow down. Or, to put it in terms you can describe in ActionScript, you want it to travel less and less every frame. You use speed to control how far the blob moves for each frame, by making it a specific fraction of the distance remaining. That figure of 10 we used to scale down the speed was actually very important. If you're adventurous, you may already have tried plugging in some different values here; it's not like we gave you any good reason to use 10 after all! Say you used 2 instead and traveled half the remaining distance in each frame. It would work just as well, only quicker.

In fact, this value directly controls how quickly the blob slows down (or **decelerates**, for the more technically minded among you. Engineers would say it controls the **stiffness** or **feedback** of the system). If it's a large number, the blob will take ages to reach its destination because the speed starts off low and falls off too quickly. If it's a very small number (between 0 and 1), the blob will overshoot the target because the speed doesn't fall off quickly enough.

How do we know all this? Well, it comes down to a bit of math beyond the scope of this book, but interested readers can do some research on control theory.

In fact, you can generate very specific behavior by carefully picking your value and testing the code. Change the code like this by removing the if condition in line 2 and its closing brace on line 5 of the existing script:

```
blob_mc.onEnterFrame = function() {
  speed = (target - this._x)/10;
  this._x += speed;
};
var speed:Number = 0;
var target:Number = 200;
```

Now try plugging in different values for 10:

- If it's less than 0.5, the blob accelerates past the target and never stops.
- If it's equal to 0.5, the blob will overshoot by exactly the original distance, overshoot back again, and so on, **oscillating** around the target forever.

- If it's greater than 0.5 but less than 1, the blob will overshoot slightly but come back and stop on the target after a few "bounces."
- If it's exactly 1 (or you simply leave it out), the blob goes straight to the target, as you saw earlier.
- If it's between 1 and 2, the blob slows down and stops at the target without overshooting.
- If it's over 2, the blob will never *quite* reach the target in theory, but because of computer rounding errors, the blob will get to the target after a second or so.

Of course, the easiest way to find a value you like is to play around with the numbers and see what happens, but these pointers will help give you some idea of what to expect. A value between 1.8 and 2 probably gives the most aesthetically pleasing result. It's fast enough without being so fast you see a jerky movement.

We can sum it up like this. For each frame, you work out the blob's new position as follows:

```
New position = position now + ((target - position now) ÷ constant)
```

Why is this important? Well, it's the simplest, most processor-friendly equation you can use to add realism to movement in your Flash movies. This is because it simulates acceleration, which suggests the blob has a whole host of real physical properties and processes acting on it (such as weight, viscosity, friction, inertia, and more).

Although this equation is about motion changes with time, it has no explicit reference to time. That's because the movie is based on frames, and it's the frame interval that creates the time-based motion. The frame rate defines how often this equation is run, and this rate defines the time taken for the journey.

You may be thinking, "Um, that still doesn't answer my question! Why is this important to Flash web design?" Good question.

Well, compare your smooth decelerating motion with the linear motion you had at the start. Not only does the nonlinear motion appear *smoother* but it's *faster* as well!

When you're designing anything that moves in Flash, from a space invader to moving parts in a cool Flash UI, you should use movement with inertia.

For a practical Flash web design application of inertia, you don't need to look very far. Take a look at the finished Futuremedia site and you'll see that when you click one of the strips, the UI zooms and fades with an inertia effect. This makes the effect both smooth and pleasing to the eye. This is, of course, why we chose inertial or "real" motion to illustrate modular programming. Both subjects are vital in Flash web design.

The Tween *class offers an alternative method of coding realistic motion. We'll take a look at that a little later in this chapter.*

You've now spent a little time looking at some fairly heavy concepts behind simulating real movement, but you haven't really tied it in with modular programming. It's surely about time you did—after all, that's what the chapter is all about! Let's make some movies to practice what you've learned so far.

Trailing the pointer (a mouse follower)

In this example, you're going to create a component that will control a movie clip and make it chase the mouse cursor. Remember the following equation:

```
New position = position now + ((target - position now) ÷ constant)
```

In this case, your target will be the mouse pointer, whose position coordinates are stored in _xmouse and _ymouse. The movie clip instance will be called trail_mc, and you'll use its properties _x and _y to maneuver trail_mc about the screen. You therefore need to use the equation twice: once for the x-coordinates and once for the y-coordinates.

```
_x = _x + ((_xmouse - _x) / n)
_y = _y + ((_ymouse - _y) / n)
```

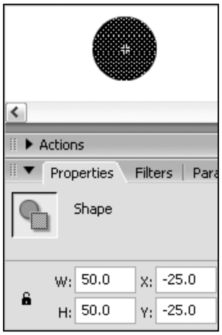
Note that you use a variable called n to denote the constant. You'll also need to remember to define that somewhere in your code.

Just in case you get stuck at any point, you can find a complete version of this exercise saved as inertia.fla in the download files for this chapter.

Creating the movie clips for the mouse follower

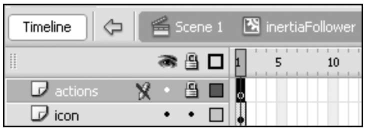
Follow these steps to create the movie clips for the mouse follower:

1. In a brand-new Flash document with the frame rate set to 24 fps, create two empty movie clips called trail and inertiaFollower by clicking the New Symbol button at the bottom left of the Library panel.
2. Open trail by double-clicking its entry in the Library and draw a black circle about 50 pixels across in the center of the stage. This will be the "blob" you use to follow the pointer. You don't need to do anything else to this movie clip—all your code will sit in the self-contained "black box," inertiaFollower.
3. Now open inertiaFollower. Add a new layer, and rename your two layers as actions and icon. Lock the actions layer.



W: 50.0 X: -25.0
H: 50.0 Y: -25.0

4. In the icon layer, draw a small red square. This is purely so you can see that inertiaFollower is there. The inertiaFollower component will consist of only code, which would otherwise make it have no graphical appearance.



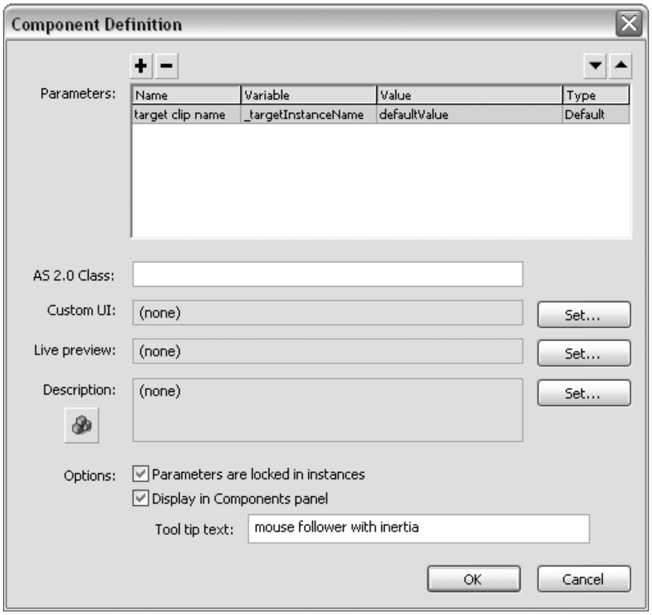
Now you need to convert inertiaFollower into a component that you can drag and drop onto objects on the stage. You also need to define a special component parameter called _targetInstanceName.

Why do you need to do this, and what was wrong with the `videocontrols` component you created (which didn't require a `_targetInstanceName` parameter)? Well, the last component wasn't completely general—it always has to be on the timeline of the target it will be controlling so that the `_parent` path is valid and points to the correct movie clip. As you'll see, `_targetInstanceName` is more general because it becomes equal to the name of whatever you drop the component onto. You don't have to worry about ActionScript hierarchies or anything else—just find the movie clip you want to control, and drop the component onto it!

> *The components you're creating here aren't structured like the ones that ship with Flash. The version 2.0 components that come with Flash 8 are class based and horrendously complicated, and they aren't normally the sort of thing you would create yourself, unless you've spent a serious amount of time understanding their structure (something that you'll begin to do in Chapter 15). The components you're creating are simple, self-contained, black-box containers used to separate code from content. They're quick to create, develop, and understand.*

5. Right-click/Ctrl-click the `inertiaFollower` movie clip in the Library, and select Component Definition. Now click the plus button (+) at the top of the Component Definition dialog box that appears. This adds a parameter in the pane below it. In the Name field, enter target clip name, and set Variable to _targetInstanceName. Make sure you spell it correctly, including the leading underscore and combination of lowercase and uppercase letters.

You can leave the other fields (Value and Type) as they are. While you're at it, select the Display in Components panel check box (at the bottom of the window to the right of Options) and enter mouse follower with inertia in the Tool tip text field.



Parameters:

Name	Variable	Value	Type
target clip name	_targetInstanceName	defaultValue	Default

AS 2.0 Class:

Custom UI: (none)

Live preview: (none)

Description: (none)

Options: ☑ Parameters are locked in instances

☑ Display in Components panel

Tool tip text: mouse follower with inertia

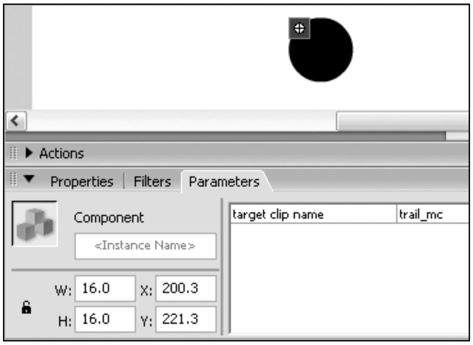
> *The* Name *field is just a bit of descriptive text that describes the parameter, so you can include spaces. However, the* Variable *field defines the actual variable, and this entry should follow the rules of variable naming.*

6. Finally, in the lower-left corner of the Component Definition dialog box, click the icon below the Description title and above Options, and select ActionScript Component icon from the pop-up menu that appears. This changes the icon that will appear in the Library panel alongside your new component. Click OK to close the dialog box.

Testing drag-and-drop functionality

Let's test the drag-and-drop feature. For this to work, you need to enable the Snap to Objects mode. You can do this by either clicking the magnet icon in the Options area of the Tools panel or selecting View ➤ Snapping ➤ Snap to Objects.

1. Place an instance of the `trail` movie clip on the stage and give it the instance name `trail_mc`. Now drag a copy of `inertiaFollower` from the Library and drop it onto `trail_mc`.
2. Take a look in the Parameters tab of the Property inspector, and you should see the following:



target clip name: trail_mc

W: 16.0 X: 200.3

H: 16.0 Y: 221.3

This means your component has read the name of the instance you dropped it on. "Um . . . so what?" you may be thinking. Well, since you know the name of the instance, *you can control it*.

> *If you don't see this, make sure you have the* `inertiaFollower` *component selected on the stage and that you're looking at the Parameters tab of the Property inspector. As shown in the previous screenshot, the panel has three tabs at the top left. (If you're using Flash Basic 8, there will be just two, since the Basic version doesn't support filters.)*

You can perhaps see that drag and drop is much more versatile than the previous videocontrols component, which used only _parent. The video controls require you to know enough about ActionScript to know what _parent is, but inertiaFollower doesn't require you to know what myTarget is—you just have to drop the component on what you want it to control and you're off and running.

> *In short, the black box for* inertiaFollower *is general, and this makes it a better modular package. A word of warning, though. It does have drawbacks, which will be explained shortly.*

Adding the inertial code

It's time for some scripting. You can't edit a component in place. The only way you can get inside it is via the version in the Library.

Open the inertiaFollower component by double-clicking it in the Library and select frame 1 of layer actions. Add the following code:

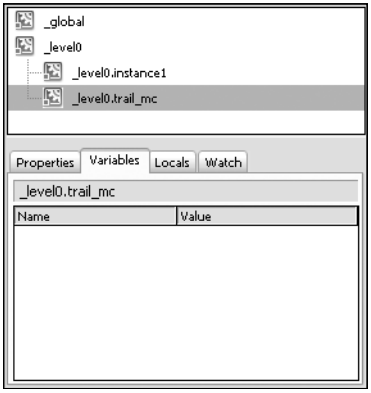
```
function inertia():Void {
  // Capture mouse positions...
  targetX = _root._xmouse;
  targetY = _root._ymouse;
  distX = targetX - myTarget._x;
  distY = targetY - myTarget._y;
  // Apply inertia equation...
  myTarget._x = Math.round(myTarget._x+(distX/5));
  myTarget._y = Math.round(myTarget._y+(distY/5));
}
// Initialize...
var myTarget:MovieClip = _parent[_targetInstanceName];
var targetX:Number = 0;
var targetY:Number = 0;
var distX:Number = 0;
var distY:Number = 0;
// set animation event...
this.onEnterFrame = inertia;
```

Although this is a fairly short bit of code, there are *lots* of little tricks here, so let's spend some time looking it over. You'll see what it does first, then you'll look at how everything is set up behind the scenes, and finally you'll examine the code itself.

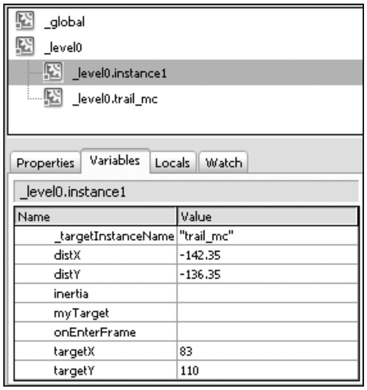
If you test this FLA (Control ➤ Test Movie), you'll see the little red icon (that is, of course, your component instance) stays where it is, whereas the ball follows the mouse. That's odd for a start, because all the code is in the square, not the circle, but we hope you've already come to expect that.

Run the movie in debug mode (Control ➤ Debug Movie). Click the Continue button in the debugger (the little green arrow) and then take a look at the Variables tab. That's where you'll see the modularity at work.

1. Select the trail_mc instance. You'll see that there's nothing there. No variables, no attached event handlers—nothing.



2. Now look inside the component (it's called instance1) and you'll see that's where the party is! The event handler is attached to the red square, and all your variables are there as well. As you can see from the screenshot, the Value of _targetInstanceName is trail_mc, giving you a direct reference to the movie clip the component is sitting on.



_level0.instance1

Name	Value
_targetInstanceName	"trail_mc"
distX	-142.35
distY	-136.35
inertia	
myTarget	
onEnterFrame	
targetX	83
targetY	110

What are the advantages of this? Well, although the trail_mc movie clip is animated by the component, *it isn't affected by the component in any other way*. This has several advantages:

- The component doesn't care what it's controlling, and the clip under the control of the component is oblivious to the component. This means that when you create the component *or* movie clip, you don't have to worry about the other. This is the black-box concept again—you can forget about the finer details.
- A consequence of the preceding point is that your component will be oblivious to *other components* dropped onto the same movie clip. This means that you can add as many components as you want to each clip. Not only does this allow you to create very complex animations by using several components, but also it allows you to create lots of individually simple components, rather than one big complicated component.

In modular design, not only are the final modules separate when you use them, but also the development of the software that forms each module can be separate. This is a major advantage of modular coding techniques.

Let's now turn our attention to how this modularity is actually implemented.

Understanding the inertial code

Let's look at the initialization first. The code sets up a reference (or path) to the target movie clip, which is _parent followed by the name of the target, _targetInstanceName. You then set up the variables you need for the inertial effect, defining the name, type, and initial value for each. Finally, you assign the onEnterFrame event handler for the animation. This event handler is *not* attached to the movie clip being animated, but to the component.

```
// Initialize...
var myTarget:MovieClip = _parent[_targetInstanceName];
var targetX:Number = 0;
var targetY:Number = 0;
var distX:Number = 0;
var distY:Number = 0;
// set animation event...
this.onEnterFrame = inertia;
```

Note the use of square brackets to enclose _targetInstanceName. *Although it looks like a property that you would normally access through dot notation, the script won't work if you use* _parent._targetInstanceName. *Strictly speaking, it's not necessary to use* _targetInstanceName *inside the component. Any variable will do. However,* _targetInstanceName *is an ActionScript component property, so its value shows up in the debugger. If you use any other variable, it won't.*

The function inertia() sets up your inertia equations and uses them to control the target movie clip. The thing to note is that you don't use this inside the function. The code is actually attached to the component's onEnterFrame event handler. The only time you need a path is when you come to animate the target movie clip (the last two lines of the function). You need to move the target, not the component, so you need to refer to myTarget._x and myTarget._y.

```
function inertia():Void {
  // Capture mouse positions...
  targetX = _root._xmouse;
  targetY = _root._ymouse;
  distX = targetX - myTarget._x;
  distY = targetY - myTarget._y;
  // Apply inertia equation...
  myTarget._x = Math.round(myTarget._x+(distX/5));
  myTarget._y = Math.round(myTarget._y+(distY/5));
}
```

There are a couple of problems with the code as it stands. One problem is that you can still see the red square that represents the component. Although it's useful when you're editing the FLA, it isn't something you want to show up in the final SWF. The other problem is that you can simply drag and drop the component, which is cool, but you can't really configure it. You can't change the inertia constant to the different values that would create a fast-moving, slow-moving, or oscillating ball, for example.

Let's look at these issues now.

Fine-tuning the component

Making the component disappear is easy. You just set the `_visible` property of the component to `false`.

1. Add the following line to the code so far. When you now test the FLA, you'll see that the red square is no longer visible. The `_visible` property is a flag that tells the Flash player "Don't draw this."

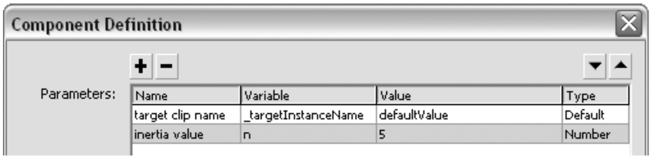
```
this._visible = false;
```

> *Setting the* `_visible` *property is the preferred way to make something disappear. Setting* `_alpha` *to a very low value isn't recommended to do the same thing. In the latter case, Flash may still be drawing your "hidden" graphics, and this can slow down your movie considerably.*

2. Next stop: adding some configurability. If you think about it, the only things that define your motion are the variables that control it. So the way to configure the inertia effect is to allow the setting of one or more variables on a per-instance basis. Here's how this is done.

   Come out of editing `inertiaFollower` and back onto the main timeline. Right-click/Ctrl-click the `inertiaFollower` component in the Library to bring up the Component Definition dialog box again.

3. Add a new parameter via the + button. Give this new parameter the name inertia value and the variable name n. Change the default value to 5, and change the Type field from Default to Number as shown in the following screenshot. Then click OK to save the new definition.



Name	Variable	Value	Type
target clip name	_targetInstanceName	defaultValue	Default
inertia value	n	5	Number

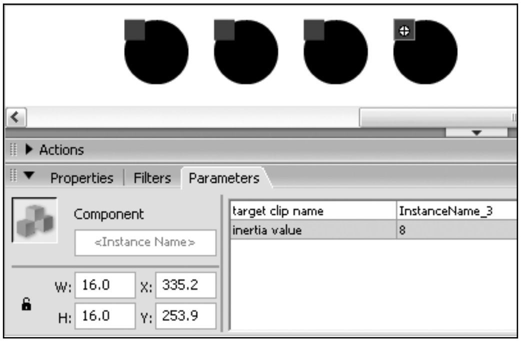
In the same way that `_targetInstanceName` was passed to the code within the component, a new variable called `n` is created every time you drag and drop an instance of `inertiaFollower` onto the stage. This variable is created inside (or with a scope of) the instance you just dragged and dropped.

4. You now need to use your new parameter in your script. Go back into `inertiaFollower` (remembering that you have to double-click the version in the Library to do this with a component). Change the following two lines in frame 1 of the actions layer, replacing `5` with `n`.

```
myTarget._x = Math.round(myTarget._x+(distX/n));
myTarget._y = Math.round(myTarget._y+(distY/n));
```

5. Now delete everything from the stage and add four instances of `trail`. You don't need to give them instance names. Flash assigns them default names.

6. Drop an `inertiaFollower` component on each circle, and use the Property inspector to give them `inertia` values of 2, 4, 6, and 8.



<Instance Name>

W: 16.0 X: 335.2
H: 16.0 Y: 253.9

target clip name	InstanceName_3
inertia value	8

7. Now test the FLA. You'll see all four circles follow the mouse. Each is controlled by a separate instance of the `inertiaFollower` component, each of which is configured with its own value of inertia.

   Finally, have a look at the download version of the FLA (if you haven't already), `inertia.fla`. We've replaced the component square with a more descriptive icon that gives a pictorial clue as to what the component does.

   By giving each instance a slightly different value, you've built an altogether more complex animation. This is all happening through modular clips and code reuse.

The techniques discussed here have to do with timeline modularity. The advantage of such structures is that you have a graphical black box in which the code is placed and modularized. The component defines all the interfaces (component parameters) via graphical interfaces—things such as drag and drop and the Property inspector. This makes using modular movie clips and components a cool way to introduce Flash modular code to designers who tend to think graphically.

There are two big disadvantages of using modular movie clips:

- **It places code all over the place**. Drag and drop is fine when you're thinking about ease of use, but when you have 20 of these cool components scattered all over a timeline, it can get difficult to think in terms of code. The code is all over the place and, although your code is still modular and well defined, there's no real overall code structure (although there is a well-defined graphic structure; the components are always on top of the thing they control).
- **The way the code is modularized isn't recognizable as modular code to non-Flash programmers**. Anyone from a JavaScript background reading the preceding sections will have had a heck of a time. This "timelines as code containers" stuff is easy for nonprogrammers to grasp, but it's tough for programmers from other fields who are new to Flash—they just don't have anything like it!

As Flash moves further toward true programming, using timelines and components in the way you've done so far is being replaced by another kind of code modularity. It is this type of modularity that you'll look at next: function-based modular code.

You need to know about function-based modular code before you can start to look at what should be your ultimate goal as an ActionScripter—class-based, object-oriented programming (OOP).

Function-based modular code

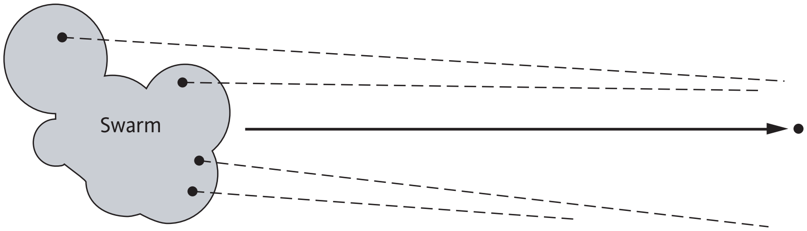
The next example may look daunting from a distance, but it's based quite closely on `inertiaFollower`, and Sham took only half an hour to put it together. Of course, he's had a bit of practice with Flash, but we guarantee that you'll be doing the same within six months if you stick with it! You can find the code in `swarm.fla`.

If you go away with one thing from this section, remember that Flash allows you to quickly visualize modular code–based changes under the control of ActionScript. Once you have the basic ingredients, it's easy to build up to a stunning, dynamic, reusable effect. This idea isn't easy to grasp, but bear with us because it's important, and once you see it, it's easy to do.

Swarming behavior

You're going to look at modeling a swarm of flies. The swarm as a whole will follow the mouse around (like in the last example), but individual swarm members have slightly different destinations. Why? Because in real life it's difficult to occupy the same place as something else! The individual members of a swarm are actually all trying to go to the same place, but they can't, and they're always trying to maintain a position in an area with limited space.

There's also the *type* of motion. A swarm in motion will tend to be spread out when it's traveling because each member needs space to fly. When it's at rest, the swarm can become tightly packed, because speeds are now much lower, and some of the members may be just stationary and hovering rather than actually moving. There's something here about the *density* of a swarm.

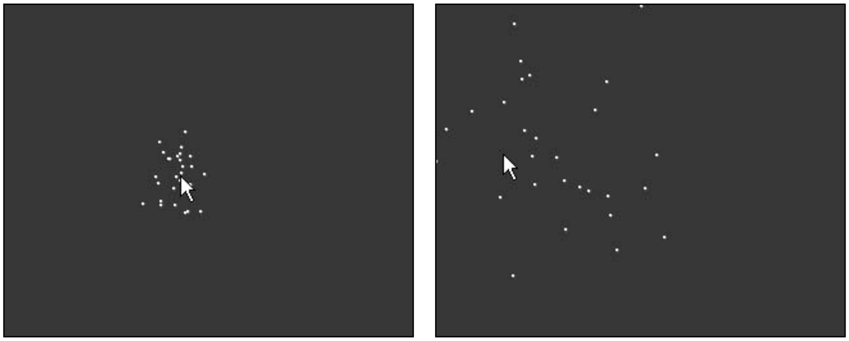


Where did all that information come from? Nowhere. No books on flies or queuing theory, chaotic motion, or fluid dynamics—just watching two `inertiaFollower` balls moving around, following the cursor. Sham combined that with a memory of walking down a country path with his partner a few summers ago and having to dodge past the odd congregation of swarming bees.

The file `swarm.fla` is very similar to `inertiaFollower` except for a few things:

- The biggest difference is that all the movie clips are controlled purely by code, and not by dropping a component onto each one.
- Two new terms, `flockX` and `flockY`, have been added to denote random positions around the cursor. So, instead of members of the swarm aiming for exactly the same target destination, it's different for each one.
- If the cursor is moving, the swarm is traveling, so the spread gets bigger. If it isn't moving, the swarm is stationary and won't be so spread out.
- The variable `oldTargetX` determines whether the cursor is moving. Only the x-coordinate is used, because if the cursor moves, it's highly unlikely that only one coordinate will change. If the current target is the same as the old target, then it's safe to assume the cursor is stationary.
- Every now and again, there's a chance that a swarm member will break away temporarily, due to its neighbors getting in the way. You therefore want to simulate this.

Although the code looks quite a lot longer, these really are the only changes that have been made. If you want to kick-start the swarm, increase the frame rate (Modify ➤ Document). The more you increase it, the angrier the swarm looks. You can almost hear the buzzing!



Open swarm.fla and test the movie. If you leave the mouse still, you'll see the swarm bunch up and flit around it as a dense swarm. If you move the mouse, the swarm gets less dense as its members move apart. Here's a happy bonus: leave the cursor still for a bit and then move it slightly. It's almost as if the swarm has been disturbed and is getting angry! It's funny how when you model simplicity, complex behaviors seem to just emerge.

When you inspect swarm.fla, you'll see that it's a one-frame timeline that contains a single script attached to frame 1. This time there are no components—everything is achieved through code. There are many advantages to creating code-heavy FLAs in this way rather than using code-containing movie clip black-boxes or components:

- All your code is in one place: the first frame of the timeline.
- Drag and drop is cool, but it has limitations. Just consider the difficulty in placing a component over every fly in your swarm! In the swarm code, the black boxes are attached to their targets using *code* rather than manually, which saves your eyesight.

> *The book* Flash Math Creativity, 2nd Edition *by a cast of thousands—well, 15 anyway—(friends of ED, ISBN 1-59059-429-0) showcases ActionScript experiments. Many of them are just exploration for exploration's sake, but many clever UI ideas spring from such tomfoolery!*

We won't go too far into this code because it's very similar to the code used in the previous example, but the swarm FLA shows how Flash can simply be **fun**. Think of a problem you want to visualize, and Flash has the power to allow you to write the code and see stuff whizzing around the stage.

You'll see the stage is completely empty, and there's just a single movie clip called bug in the Library. Open the Actions panel and take a look at the script. Everything is controlled by two functions: animate() and initialize(). As their names suggest, animate() controls the movement of each movie clip, while initialize() sets the initial values for each movie clip's timeline variables, and assigns animate() as the movie clip's onEnterFrame event handler. The main code is at the bottom of the listing (line 39 onward):

```
var i:Number = 0;
var bugClip:MovieClip;
for (i = 0; i < 30; i++) {
  bugClip = this.attachMovie("bug", "bug"+i, i);
  initialize.apply(bugClip);
}
```

The loop creates 30 movie clips, bug0 to bug29. The last line of the script applies the initialize() function to the current movie clip, whose name is stored in the movie clip reference (or "path to the movie clip"), bugClip. The *functionName*.apply(*target*) method of the Function class creates a copy of the function function on the timeline of the movie clip target.

Hang on a minute. That's does exactly the same as the drag-and-drop component earlier! With components, you physically place a "code container" on the target movie clip. When the SWF runs, the component sets up the code structures that animate the target.

This time, the code container is a function—initialize(). The black-box concept has been generalized to a much greater degree. There's no longer a physical (rather, *visual*) box for the code. Instead, this uses ActionScript directly to define the black boxes using code only. Here's the complete code listing for swarm.fla:

```
function animate():Void {
  // Capture mouse positions and distance from mouse
  this.targetX = _root._xmouse;
  this.targetY = _root._ymouse;
  this.distX = this.targetX-this.meX + this.flockX;
  this.distY = this.targetY-this.meY + this.flockY;
  if (this.targetX == this.oldTargetX && Math.random() > 0.9) {
    // add small scale random darting if mouse is still
    this.flockX = (Math.random()*100)-50;
    this.flockY = (Math.random()*100)-50;
  } else if (this.targetX != this.oldTargetX && Math.random() > 0.8) {
    // add large scale random darting if mouse is moving
    this.flockX = (Math.random()*400)-200;
    this.flockY = (Math.random()*400)-200;
  }
  // Apply inertia equation
  this.meX = Math.round(this.meX + (this.distX)/20);
  this.meY = Math.round(this.meY + (this.distY)/20);
  // perform animation
```

```
    this._x = this.meX;
    this._y = this.meY;
    // remember mouse pos so we can tell if it has moved next time around
    this.oldTargetX = this.targetX;
  }
  function initialize():Void {
    this.cacheAsBitmap = true;
    this.targetX = 0;
    this.targetY = 0;
    this.distX = 0;
    this.distY = 0;
    this.meX = this._x;
    this.meY = this._y;
    this.oldTargetX = 0;
    this.flockX = (Math.random()*200)-100;
    this.flockY = (Math.random()*200)-100;
    this.onEnterFrame = animate;
  }
  // Create the movie clips and initialize them
  var i:Number = 0;
  var bugClip:MovieClip;
  for (i = 0; i < 30; i++) {
    bugClip = this.attachMovie("bug", "bug"+i, i);
    initialize.apply(bugClip);
  }
```

Although you could do this via embedded timelines and components, you'll tend to find that your code ends up very fragmented—split across all sorts of timelines. This way, you see the full "Russian-doll" hierarchy in a single script.

Let's review the more advanced advantages of the new, function-based code before moving on:

- It makes code **target independent** (that is, you don't need to know what the thing you're controlling is called when you write your code; you simply use the general path "this," as in "this bug movie clip").
- It makes code **position independent** (so you don't need to know where your code will end up). Nested timelines that use the `_parent` trick are only target independent. They still rely on position because they have to be the child of the clip they're controlling. The code in `swarm.fla` is *totally* independent of the application you wind up using it for, and that makes it more reusable. You could take the script and just place it into another FLA, change the main loop at the end slightly to refer to some other movie clips, and the whole thing would work.
- You have more control over the final effect. For example, change the `i < 30` part of the `for` loop at the end of the script to 200, and rerun the script. Now think how long it would take you to do the same thing manually. The way you've written the code makes it easier to tweak because all the controls are software parameters rather than manual "put 50 of these on the stage" deals.

- You can define event handlers *before* the movie clip you're applying them to exists. If you think about it, you've defined an onEnterFrame handler without having anything to attach it to. That makes for flexibility. You can choose to *not* attach the swarm event, or you can allow the user to control the effect and select from different events at runtime. For a good example of this, change the for loop at the end of the script as follows:

```
this.onMouseDown = function() {
  var i:Number = 0;
  var bugClip:MovieClip;
  for (i =0; i < 30; i++) {
    bugClip = this.attachMovie("bug", "bug"+i, i);
    initialize.apply(bugClip);
  }
  delete this.onMouseDown;
};
```

  This won't attach all the script to the instances (in fact, it won't even create the instances) until you click the mouse. You've taken the generation of script away from a linear path and made it **conditional**. This not only leads to increased flexibility, but also allows more complex structures to be built that support cool things like adaptive behavior.

Taking the swarming effect forward

Let's say you want to develop the code further. Instead of swarms chasing the cursor, you want them chasing food. The food would be a plant called "Spore" growing on a distant planet. The user could cause Spore to grow or die by changing the climate.

The functions that control the swarm members aren't fixed. They can mutate in response to the changing conditions. The members can be grazers instead of flies that eat Spore; or they could evolve to be stronger and eat other grazers; or they could become scavengers, eating Spore but keeping an eye on the weaker grazers.

There are all sorts of other possible code-based stuff you could add via additional modules like pollution (a function that causes swarm elements to die if they come too near a new kind of movie clip called a pollutant), growth (the attributes of a grazer change over time to represent a born-live-die cycle), and aggression (variables define how tenaciously a swarm member tries to get to its target). You might well call this game "Ecology," and it stops being a trick FLA and starts to become something useful and fun to kick around. In fact, in the next chapter, you'll see how we have adapted the code to create a swarm of hostile aliens in a version of the classic Space Invaders game.

OK, that's enough to whet the appetite for future days. Let's look at something else for today and *now*.

Creating tweens with ActionScript

Since we've been discussing realistic movement, let's take a brief look at the Tween class. Yes, that's right: you can use ActionScript to create motion tweens.

The Tween *class works in Flash Player 6 and later, so you can safely use it in all your movies unless a vital part of your target audience still clings to Flash Player 5 (roughly one percent of all Flash users as of late 2005). You can also use the* Tween *class even if you're still using Flash MX 2004. It was there all the time, but it wasn't formally documented until Flash 8.*

As you might expect, the Tween class replicates the same sort of effect as a motion tween using the timeline, but it has the following additional advantages:

- It offers a wider range of effects, including bouncing and elastic movement.
- Duration can be based on either frames or seconds elapsed.
- Tweens can be applied to dynamically generated movie clips.
- The onMotionFinished event handler allows you to assign a function that will be triggered as soon as the tween comes to an end.

Using import to access the Tween class

Although the Tween class is included in Flash 8, it's not part of the core ActionScript, but belongs to the part of the language used to control Flash version 2 components. So you need to tell Flash where to find the necessary code. It's just a simple matter of putting the following commands at the top of the Actions panel in any movie where you want to use ActionScript to create a tween:

```
import mx.transitions.Tween;
import mx.transitions.easing.*;
```

Both of these two lines use the import keyword, which is new to ActionScript 2.0. You use import to tell the Flash compiler where to find classes that aren't built into the core ActionScript language, so that it can compile the necessary code into your SWF. Don't confuse import with the #include command, which copies external code into the current script, even if the code isn't needed in your final movie. The import command simply tells the compiler where to find code if it needs it at compile time. It is only ever used for ActionScript 2.0 classes, which are covered in more detail in Chapter 15.

You've probably noticed that the second import command ends with an asterisk (*). As you might expect, this is a wildcard character that tells Flash to access all classes within the mx.transitions.easing **package**. (A package is a set of ActionScript 2.0 classes stored in a common location.) Since Tween is also in mx.transitions, you might expect to be able to do this:

```
import mx.transitions.* // Don't use this!
```

The dot notation in package names indicates the folder (directory) structure that classes are stored in. So easing is actually the name of a folder. The wildcard character imports only classes. It can't be used with folder names.

How to use the Tween constructor function

The Tween constructor function takes seven arguments, all of which are required. Yes, that's right—*seven* arguments! Fortunately, they're easy to remember:

- The instance name of the movie clip you want to apply the tween to
- The movie clip property that you want to tween (for instance, _x or _alpha)
- The type of tween (Flash calls it an easing class and method.)
- The starting value of the tween (for instance, an x-coordinate on the stage or an _alpha value)
- The final value of the tween (where you want it to end up)
- Duration (in either seconds or frames)
- A Boolean value that tells Flash how the duration is specified (true for seconds, false for frames)

What might catch you out is that the name of the movie clip shouldn't be enclosed in quotes, whereas the name of the property must be. Things should become clearer with an example.

Experimenting with tweens

1. Open the FLA containing blob_mc that you created earlier in the chapter and save it as tween1.fla. Alternatively, use tween1.fla from the download files for this chapter.
2. Select frame 1 on the actions layer, open the Actions panel, and replace the existing ActionScript with the following code (it's already there in the download version of tween1.fla):

```
import mx.transitions.Tween;
import mx.transitions.easing.*;
var myTween:Tween = new Tween(blob_mc, "_x", Strong.easeOut, ➥
10, 200, 3, true);
```

Let's go through each of those arguments in turn:

- blob_mc: This is the instance name of the movie clip the tween is being applied to.
- "_x": This is the property of blob_mc that you want to tween. Remember, the property must always be in quotes.
- Strong.easeOut: This specifies the type of tween, and always consists of two expressions joined by a period—the easing class and method you want to use. We'll explain them in detail in a moment.
- 10: This specifies the start value of the tween. Since it's being applied to the _x property of blob_mc, the tween will start from an x-coordinate of 10 on the stage.
- 200: This specifies the final value of the tween. So blob_mc will finish up at an x-coordinate of 200 pixels.
- 3: This sets the duration of the tween.
- true: This tells Flash that you want the duration measured in seconds. So this tween will last three seconds. If you use false, the tween will last three frames instead.

> *If you entered the code yourself, you will have noticed that Flash doesn't provide any code hints for* Tween. *If you need a reminder of the correct order of arguments, press F1 to open Flash Help and select* Components Language Reference ➤ Tween Class ➤ Using the Tween class. *The same page contains full details of the types of tween available.*

3. Test the movie to see the effect. It's almost identical to the inertia formula you used in `blob2.fla`. The blob accelerates across the stage and comes to a gradual halt.

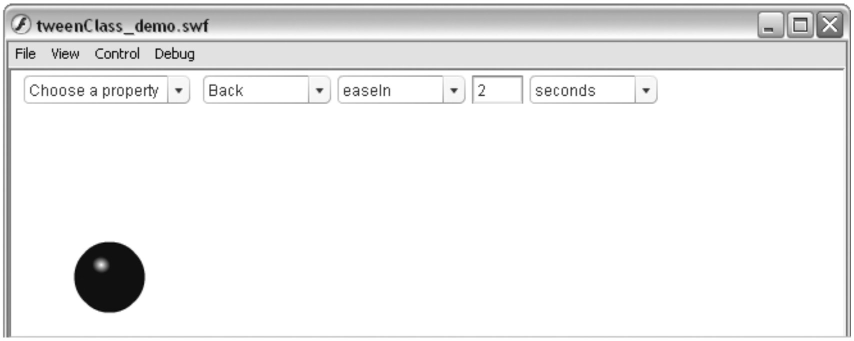
4. Open the Actions panel and change the third argument from `Strong.easeOut` to `Strong.ease`**`In`**`Out`. If you test the movie again, you'll see that the blob starts off slowly, accelerates, and then comes to a gradual halt. Achieving this type of movement would be much more complicated if you attempted to use a mathematical formula.

5. Now change the third argument from `Strong.easeInOut` to `Strong.`**`easeIn`**. This has the effect of starting the blob slowly and, once full speed has been attained, maintaining it to the end of the motion.

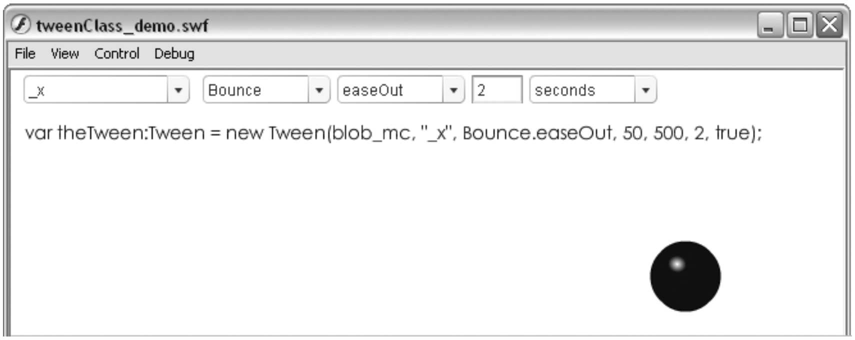
Understanding the transition types

Now that you've seen the effects of `easeOut`, `easeInOut`, and `easeIn`, it's a good idea to experiment with some of the available transition types. There are six in all: `Back`, `Bounce`, `Elastic`, `Regular`, `Strong`, and `None`. This last one is only ever used in combination with easeNone, and performs the type of linear transition with no inertia effect that you saw in `blob1.fla`.

To make things easier for you, we've created a file called `tweenClass_demo.fla`. There's no need to add anything to the file. Just publish it and play with the SWF. When you open `tweenClass_demo.swf`, you will see a series of drop-down menus, a text input field, and a stationary ball, as shown in the following screenshot:



The drop-down menus let you experiment with the effect of the various easing classes and methods on the `_x`, `_y`, and `_alpha` properties of the ball, which has an instance name of `blob_mc`. The text input field also lets you change the duration, which can be toggled between seconds and frames. Every time you change one of the values, the movie applies the equivalent tween to blob_mc and displays the code it used to do so.



_x | Bounce | easeOut | 2 | seconds

```
var theTween:Tween = new Tween(blob_mc, "_x", Bounce.easeOut, 50, 500, 2, true);
```

Because None and easeNone can only be used together, selecting either of these values will automatically set the other drop-down menu to the appropriate setting. To use any of the other easing classes or methods again, reset the first drop-down menu to its default value: Choose a property. You can then freely change the values of the second and third drop-down menus before selecting another property to experiment with.

Use the same technique if you want to change the duration by more than one digit. The duration text input field will accept only positive numbers. If you insert a decimal fraction for frames, the SWF automatically rounds it up to the next whole number. A tween with a duration of 0 will not execute. Extremely short durations will execute, but they may make the movie unstable.

Although seeing the easing classes and methods in action is the best way to get a feeling for how they work, the following tables summarize the effects they create:

Easing class	**Description**
Back	Gives the impression of the movie clip being pulled back at one or both ends of the tween.
Bounce	Self explanatory. The longer the duration, the greater the number of bounces.
Elastic	Very similar to Back, except that the movement reverberates more.
Regular	Smooth transition with slower movement at one or both ends.
Strong	Same as Regular, but more pronounced.
None	No effects, acceleration or slowing down. Used only in combination with easeNone.

Method	Description
easeIn	Easing effect at the beginning of the tween
easeInOut	Easing effect at both ends of the tween
easeOut	Easing effect at the end of the tween
easeNone	Used only as None.easeNone

One of the advantages of using ActionScript to create tweens is that you can use the onMotionFinished event handler to schedule another tween (or any other event) to execute immediately after the previous one has been completed. The file tweens.fla in the download files for this chapter shows a simple example of several tweens being applied to the same movie clip in succession, using onMotionFinished.

Repeating a tween

1. Continue working with the file from the previous exercise or use tween2.fla from the download files.
2. Open the Actions panel on the first frame of the actions layer, and insert the following code at the end of the existing script:

```
myTween.onMotionFinished = function() {
  myTween.yoyo();
};
```

3. Test the movie. Depending on which type of tween you are using, the blob will bounce back and forth across the screen performing the same tween over and over again. However, there's a big difference between what's happening here and a normal timeline motion tween. A timeline tween always goes in the same direction, whereas the aptly named yoyo() method of the Tween class switches the direction each time.

Although using yoyo() to bounce a blob endlessly back and forth across the stage is likely to induce homicidal tendencies in visitors to your site, you could use it more effectively with a more subtle tween, such as gradually fading a movie clip in and out.

In most circumstances, onMotionFinished would probably be used to handle a series of actions wrapped in a function. For instance, you could use the Tween class to fade in some text as part of a title sequence, and use onMotionFinished to load some new text to which another tween is applied. Then, at the end of the sequence, onMotionFinished could be used to load the main movie clip. However, if you simply want to use the yoyo() method, you can assign it directly to onMotionFinished like this:

```
myTween.onMotionFinished = myTween.yoyo;
```

Leaving off the parentheses at the end of the method or function name has the effect of assigning the method or function to the event handler, which automatically runs it whenever the event is triggered. You will see more examples of this sort of assignment in the next section, which starts adding some real actions to the Futuremedia site.

This has been only a brief introduction to the wonderful world of ActionScript tweens. The beauty of the Tween class is that you can use it to perform transitions on just about any movie clip property, not just the three demonstrated here, making it far more versatile than timeline motion tweens. Although a tween can handle only one property, there's nothing to prevent you from nesting two or more tweens inside a function to create a more complex effect. In *Flash 8 Essentials* (friends of ED, ISBN 1-59059-532-7), Paul Barnes-Hoggett demonstrates how to use three separate tweens in parallel to control color transitions in the red, green, and blue channels of two movie clips.

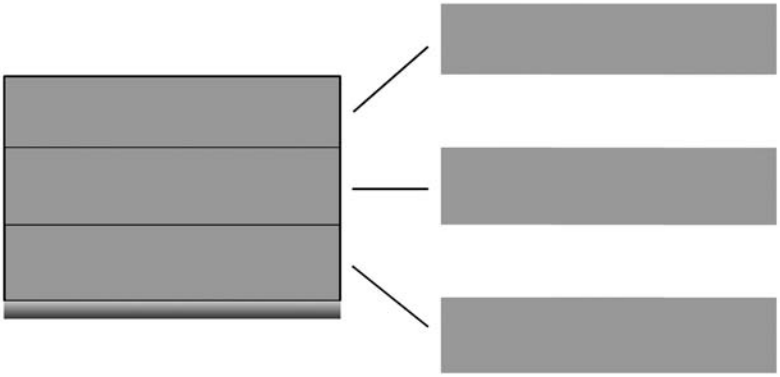
Book project: Setting up the color transition

In Chapter 7, you started adding ActionScript to the Futuremedia site, but nothing moves yet and the `tricolor` movie clip is still a dull gray. All that's about to change. Either continue working with your own file or use `futuremedia_initialize.fla` from the end of Chapter 7. The finished FLA for this chapter is called `futuremedia_code01.fla`. Feel free to refer to it if you get stuck or need something to compare with yours.

Making it modular

So, after going through the earlier parts of the chapter, you know that modular code in all its various forms is good, but what do you need to modularize for Futuremedia? This can sometimes be a difficult question for the designer learning how to program, but Flash helps you out here because it's arguably the easiest application for design/visual-based people to use when learning advanced code. Why? Because, as you've seen, Flash is a visual environment.

Your graphics are already arranged in a visual hierarchy, and that is your clue to how you need to modularize your code. You have a single clip containing three smaller clips (`tricolor` and the three strips that follow), and you'll need to reflect this hierarchy in the code.



The code associated with the strip and tricolor functionality will get quite involved, so it's better to separate it out into functions that you can write separately. You'll call the function that sets the tricolor page up (unsurprisingly) stripPage(). Your code will be on the main timeline, _root. The problem is how to control the movie clip tricolor with your soon-to-be-written function stripPage(). There are two ways to do this in ActionScript:

- Make stripPage() an event handler of tricolor. This enables you to scope tricolor via this.
- Use Function.apply() to change the scope of stripPage() so that it scopes tricolor instead of the main timeline of the current FLA, _root (or _level0).

Let's go for the second option. Why? Because you want to use stripPage() to simply initialize tricolor. You don't want it to do anything interactive or animation-specific, which is what the first option (using event handlers) is for.

There are two main things the function stripPage() needs to do:

- Define the code that *sets up* the ability to change color, and also include the *ability* to change the colors. You have to *initialize* this ability as well as *include* this ability.
- Define the event handler scripts that change position. As you'll see later in the case study, there isn't just one event handler to perform the animation, but a sequence of them, each handling a particular part of the animation before handing over control to the next event handler in the sequence. You'll start the ball rolling by adding the first one in this sequence, navigate().

You could make stripPage() a massive piece of code that does all your code setup, but there's a much easier way to keep all your code manageable: make stripPage() refer to two other functions in a modular fashion. These two functions will encapsulate the actions described earlier.

The problem with calling two new functions is *speed*. It's a bit like trying to talk to someone else through a third person because you don't speak the same language. It would be far better if you learned the other person's language, and then you could just say "Hi!" directly. The function-based version of this is assignment. Rather than *call* another function, you can make stripPage() *assimilate* these other functions into itself. That's cool because you can still write the functions separately, but at runtime you just call stripPage(), and stripPage() contains the other functions you need without having to go and ask.

Setting up the basic controls for the page

1. Continue working with your own file, or use futuremedia_initialize.fla from the download files for Chapter 7. Type in the following code below the definition of the existing navStrip variables:

```
// Initialize UI...
stripPage.apply(tricolor);
```

2. You now need to write the stripPage() function and the two functions that it refers to. Call these two functions stripEvents() and stripCols(). The temptation is to put the new code *after* the code you have so far, but the convention is to define all functions at the top of your code. So, insert the following code *before* the main code you've written so far:

```
// DEFINE MAIN SETUP SCRIPTS
function stripEvents():Void {
  // attach button events to the 3 pages...
}
function stripCols():Void {
  // set the colors for each strip
}
function stripPage():Void {
  // note use of Function assignment
  this.inheritEvents = stripEvents;
  this.inheritCols = stripCols;
  this.inheritEvents();
  this.inheritCols();
}
```

At the moment, stripEvents() and stripCols() contain just comments—you'll fill in the code in a moment, but first let's take a close look at what happens inside stripPage().

Sanity check #1

Although the stripPage() function is a deceptively short bit of code, there are a few things to notice here. Some of them are revisions to general modular code techniques you've already looked at in the book so far, but some of them are a new slant on old ideas and concepts. Let's go through them all, given that you need to know them to really understand the Futuremedia site, and some are easy to miss.

- Assigning a function is written differently from calling it. You leave out the parentheses at the end of both function names. This line of code means "Make myInheritedFunction() the same as another function called myFunction()":

  ```
  myInheritedFunction = myFunction;
  ```

- The following code is saying, "Run myFunction() and make me equal to the result it returns (if any)":

  ```
  myReturnedValue = myFunction();
  ```

- Although you saw the Tween class yoyo() method assigned to onMotionFinished earlier in the chapter, assigning a function or method doesn't mean that it's executed immediately. It's like using a variable—the values (or actions) are stored for future use. When a function or method is assigned to an event handler, it doesn't run until the event is triggered. In the case of stripPage(), the two functions called stripEvents() and stripCols() are assigned to other functions. Since there is no event handler to run them, you need to run them directly inside stripPage().
- The implied target of this is important. Because stripPage() is applied to the tricolor movie clip, references to this inside stripEvents() and stripCols() refer directly to tricolor.

Bringing color and movement to the strips

1. Let's now fill in the details of the two functions assigned to stripPage(). First, stripEvents(). Add the code highlighted in bold, as follows:

```
function stripEvents():Void {
  // attach button events to the 3 pages...
  this.strip0_mc.onRelease = navigate;
  this.strip1_mc.onRelease = navigate;
  this.strip2_mc.onRelease = navigate;
}
```

This script assigns a function called navigate() (which you'll define later) as the onRelease event handler to your three mc.strip instances inside the bigger tricolor instance. Notice that you again omit the () at the end of the function. Because you're attaching navigate() to an event, the function will be run automatically each time the event occurs.

The second thing to notice about the stripEvents() function is how it's referring to the individual instances inside the tricolor movie clip. Although the this of stripEvents() is tricolor, look what happens next: you have three lines that refer to the embedded instances this.strip0_mc to this.strip2_mc. What you're saying here (through the code) is that inside this (tricolor) you'll find another movie clip called strip0_mc (or one of the other two). The *really* cool thing is when you look at the full line:

```
this.strip0_mc.onRelease = navigate;
```

Any reference to this inside navigate() will no longer refer to tricolor, but to the movie clip strip0_mc (or strip1_mc or strip2_mc) that it has been assigned to as the onRelease event handler.

As your functions work back from the first function, stripPage(), they're fanning out to cover more and more of the stuff that needs defining. You're following the exact same hierarchy as the movie clips, with the strips inside tricolor; but instead of relying on the graphics, you're creating this structure via code, through things like inheritance and scope.

2. The code for stripCols() is a little more complicated. First, add the new lines highlighted in bold, as follows:

```
function stripCols():Void {
  // set the colors for each strip
  this.colStrip0 = new Color(this.strip0_mc);
  this.colStrip1 = new Color(this.strip1_mc);
  this.colStrip2 = new Color(this.strip2_mc);
  this.setCol = function(col0:Number, col1:Number, col2:Number):➥
Void {
    this.colStrip0.setRGB(col0);
    this.colStrip1.setRGB(col1);
    this.colStrip2.setRGB(col2);
  };
}
```

If you look carefully, the stripCols() function does something strange: it defines a function *within itself* called setCol(). It's rather like those nested Russian dolls, where you open the big one to find a smaller one inside, and another smaller one inside that, and so on. This is another structure in your code that's following the tendrils of your problem up through the trunk and out to the branches. Your initial function has to apply itself only to the single trunk, and the code will split and redefine itself as it encounters the problem branching out.

So let's take a closer look at what it actually does. As you learned way back in Chapter 2, the only graphics that can have their colors changed dynamically by ActionScript are movie clips. What may come as a surprise, though, is that there is no color property in the MovieClip class nor does the class include any methods to set the color of a movie clip. Instead, working with colors involves a two-stage process:

- Create a Color object with a movie clip as the target.
- Apply methods of the Color class (such as setRGB()) to the object you have just created.

To see how this works, take a look at the following line from stripCols():

```
this.colStrip0 = new Color(this.strip0_mc);
```

Inside stripCols(), this refers to the tricolor movie clip, so the preceding line of code creates a new property of tricolor called colStrip0. By passing this.strip0_mc as an argument to new Color(), you can use colStrip0 to manipulate the color of strip0_mc. Since strip0_mc is a movie clip in its own right, even though it's nested inside tricolor, colStrip0 controls the color of strip0_mc quite independently of the other two strips, which have Color objects of their own: colStrip1 and colStrip2.

Although you're defining new Color *objects here, notice that the code doesn't use* var. *Nor does it declare the type of* colStrip0, colStrip1, *or* colStrip2. *This is because they are being assigned as properties of the* tricolor *movie clip, and not as local variables. If you attempt to use* var *or typing with properties, you will generate a compilation error.*

Right, that's the first stage of the process out of the way. To change the color of strip0_mc, you need to apply the setRGB() method to its associated Color object, in other words to tricolor.colStrip0. The setRGB() method takes a number, which is normally a hexadecimal between 0x000000 (black) and 0xFFFFFF (white), representing the new color of the movie clip. (You can use a decimal integer, but it must be the equivalent of the six-digit hexadecimal number.)

So, the nested setCol() function takes three arguments, all numbers, and passes them to the relevant Color object for each strip. The following line sets the color for strip0_mc:

```
this.colStrip0.setRGB(col0);
```

Remember, this still refers to tricolor at this stage, so this line is the equivalent of

```
tricolor.colStrip0.setRGB(col0);
```

There's one final subtlety you need to be aware of. When you call stripCols() from stripPage(), only the following lines are executed:

```
this.colStrip0 = new Color(this.strip0_mc);
this.colStrip1 = new Color(this.strip1_mc);
this.colStrip2 = new Color(this.strip2_mc);
```

In other words, the properties that control the color of the three strips are created. However, the function setCol() is then *defined* but not *executed*. The rationale for this is obvious if you step back and think about it. The stripPage() function is called just once, when the movie first loads. It sets everything up, including the definition of setCol(). But you need to be able to use setCol() every time you navigate through the site and change the color of each strip.

If you're having trouble with this, think of stripCols() as a two-stage rocket. When you launch (initialize) the rocket, you want only the lower first stage to burn (execute). Once the rocket is up in orbit, you want the smaller, second stage to actually do stuff, and that part is called setCol().

Your color functionality is now complete. Let's put it into action.

3. The three strips inside tricolor are still a dull gray when the movie first loads, so you need to call setCol() as part of the main script. Add the code highlighted in bold to the very bottom of your listing:

```
// Initialize UI...
stripPage.apply(tricolor);
// Set UI colors...
tricolor.setCol(color0, color1, color2);
// now sit back and let the event scripts handle everything!
stop();
```

The values for color0, color1, and color2 were set in the code that you added in Chapter 7, so this will finally bring some color to the movie. What about movement? We'll start work on that in earnest in the next chapter, but let's start by laying the foundations.

4. The stripEvents() function, which you created in step 1, assigned an as yet undefined function called navigate() as the onRelease event handler for each strip. Before getting down to writing the script for navigate(), it's a good idea to check that you're on the right track by making sure that your ActionScript recognizes which movie clip you have clicked. So let's test it with trace(). Put the following code right at the top of the Actions panel above all existing code:

```
// DEFINE EVENT SCRIPTS
function navigate():Void {
  // initialize navigation event handler here
  trace("");
  trace("you pressed me and I am");
  trace(this._name);
  trace("but you need to fully define me");
  trace("before I start doing anything else!");
}
```

5. OK, you're ready to test the FLA. Did it work?

Sanity check #2

With a script as long as this, there are bound to be errors, so if it doesn't work as expected, here's a quick run-through of potential problems:

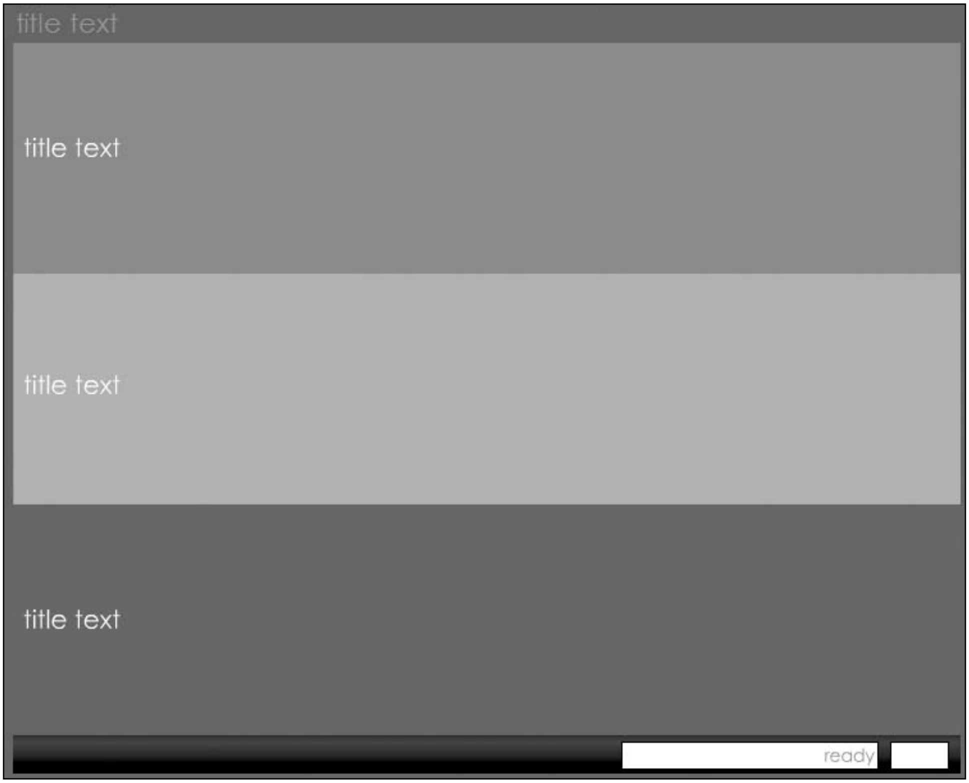
- Check that all your instance names are correct on the movie clips tricolor and strip0_mc, strip1_mc, and strip2_mc. If you used futuremedia_initialize.fla as your starting point, you don't have to worry about this.
- Check that all the variable and function names are correct. A good way to do this is to use the Find icon, which is the second icon from the left at the top of the Actions panel. If searching for a variable doesn't stop at every line you expect it to, then the missing line has a typo.
- Don't forget that ActionScript 2.0 is case-sensitive—stripEvents() and stripevents() are treated as two different functions.

Finally, here's the code listing you should have so far:

```
// DEFINE EVENT SCRIPTS
function navigate():Void {
  // initialize navigation event handler here
  trace("");
  trace("you pressed me and I am");
  trace(this._name);
  trace("but you need to fully define me");
  trace("before I start doing anything else!");
}
// DEFINE MAIN SETUP SCRIPTS
function stripEvents():Void {
  // attach button events to the 3 pages...
  this.strip0_mc.onRelease = navigate;
  this.strip1_mc.onRelease = navigate;
  this.strip2_mc.onRelease = navigate;
}
function stripCols():Void {
  // set the colors for each strip
  this.colStrip0 = new Color(this.strip0_mc);
  this.colStrip1 = new Color(this.strip1_mc);
  this.colStrip2 = new Color(this.strip2_mc);
  this.setCol = function(col0:Number, col1:Number, col2:Number):Void {
    this.colStrip0.setRGB(col0);
    this.colStrip1.setRGB(col1);
    this.colStrip2.setRGB(col2);
  };
}
function stripPage():Void {
  // note use of function assignment
  this.inheritEvents = stripEvents;
  this.inheritCols = stripCols;
  this.inheritEvents();
  this.inheritCols();
}
// Define screen extents for later use...
Stage.scaleMode = "exactFit";
var BORDER:Number = 30;
var BOTTOM:Number = Stage.height-BORDER;
var TOP:Number = BORDER;
var RIGHT:Number = Stage.width-BORDER;
var LEFT:Number = BORDER;
Stage.scaleMode = "noScale";
// Define initial screen colors
// (further colors will use these colors as their seed)
var color0:Number = 0xDD8000;
var color1:Number = 0xAACC00;
var color2:Number = 0x0080CC;
// Define navigation strip variables...
var NAVSTRIP_X:Number = BORDER;
var NAVSTRIP_Y:Number = BOTTOM;
// Initialize UI...
stripPage.apply(tricolor);
// Set UI colors...
tricolor.setCol(color0, color1, color2);
// now sit back and let the event scripts handle everything!
stop();
```

Running the FLA: The results

Here's what you'll see if all has gone according to plan:



First off, you have color! The three color variables, `color0` to `color2`, have been applied to the three strips `strip0_mc` to `strip2_mc`. You can play around with the values you've equated with the `color0`, `color1`, and `color2` variables if you wish. For example, you can design alternative colors in the Color Mixer and replace one or more of the values used in the code here:

```
// Define initial screen colors
// (further colors will use these colors as their seed)
var color0:Number = 0xDD8000;
var color1:Number = 0xAACC00;
var color2:Number = 0x0080CC;
```

For example, try changing `color2` as follows:

```
var color2:Number = 0xAAAA22;
```

This will give you yellow instead of blue. Change `color2` back to its original value when you're finished, so you're building the same site. Next, mouse over any of the three strips, and the mouse will turn to a hand icon. This means that the strips are now acting as buttons. If you click any of them, you'll see the following text appear in the Output window:

```
you pressed me and I am
strip0_mc
but you need to fully define me
before I start doing anything else!
```

The name will change depending on whether you click strip0_mc, strip1_mc, or strip2_mc. This shows that the site is aware of where you're asking to go, because each strip returns its own name.

Parting shots

You may not think you're very close to a fully working site here, but you have a lot of important features covered in the code so far.

- You've set up the variables that tell Flash what the UI looks like and where all the important points are on the screen. If you ever need to change the stage size or position of any of the major parts (something you'll have to do when it's time to customize the site for your own purposes), you know that these will be the only variables you'll have to redefine. This makes your design *flexible*.
- You've defined a structured code hierarchy early on in the project. This makes your code easy to *modify*.
- You've initialized the main functions that define your UI and attached all the main events.

You may be wondering why we didn't use the Tween class to achieve the color transitions. There are two reasons. First, there is often more than one way to achieve the same effect. Secondly, a lot of transitions are going on at the same time, and they all need to be coordinated. Since a separate tween needs to be set up for every property that's being changed, it's more efficient to handle them the way we've shown you.

You'll complete the site in later chapters, but for now, it's a good idea to play around with the code for a while using the debugger to see where everything actually ends up. As you move forward, the basic structure of the code will start to get more difficult to see as you add more lines, so now is a good time to familiarize yourself with it.

Another point worth mentioning at this stage is that this is *not* a beginner site. Neither is it a super-advanced code-fetish site that uses all sorts of highbrow code for the sake of it.

You're more than halfway through the book, and you're no longer a beginner. On the other hand, you don't want to go off creating more code than you have to. We assume that all readers of this book want to make some money out of Flash design at some point, and code for code's sake is a bad bet for profitability!

The Futuremedia site you're developing is the sort of thing you would be expected to come up with if you set your mind to creating an ActionScript site that someone else was paying you to create. In short, it's a real site that will stand up to commercial scrutiny.

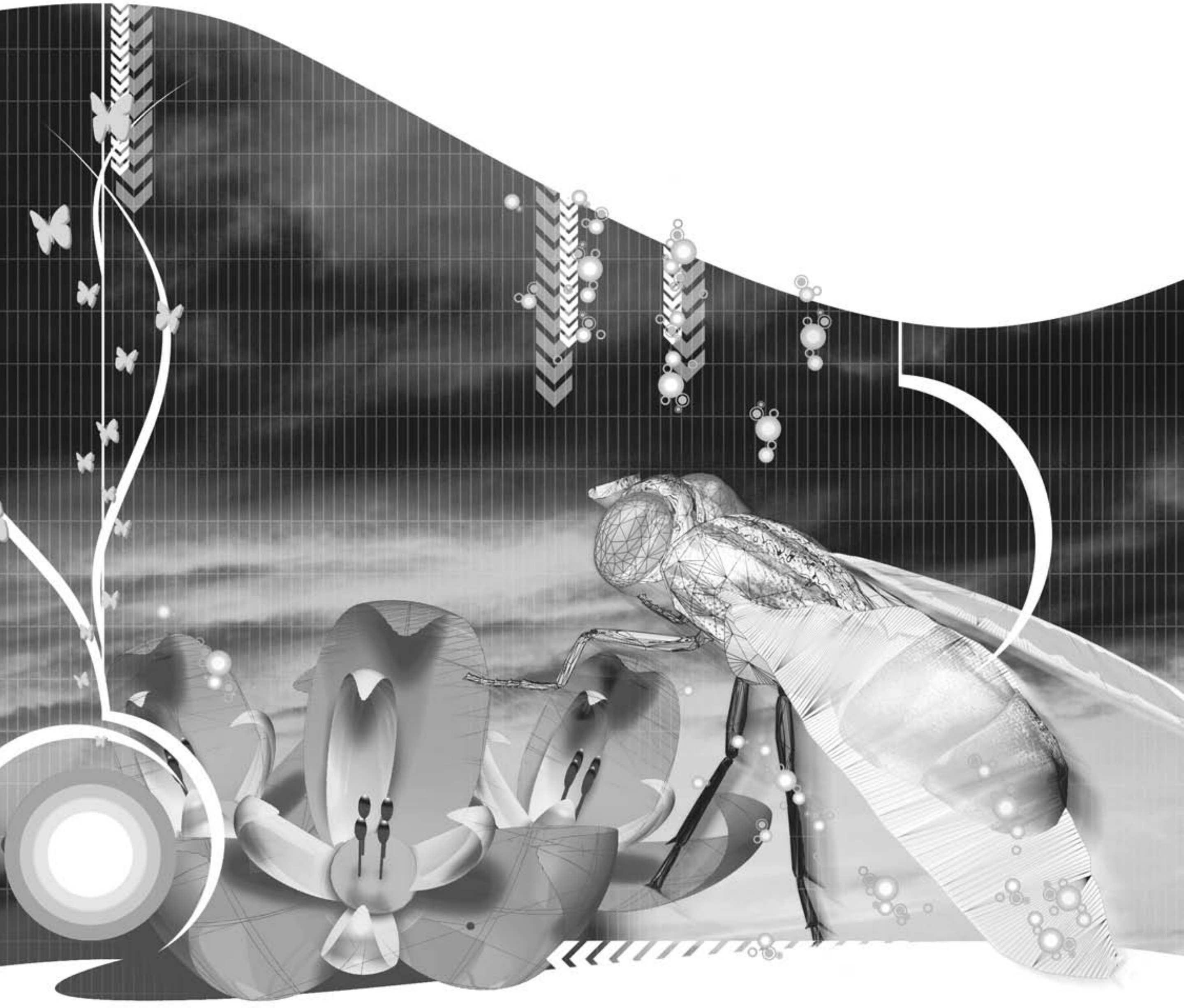
Summary

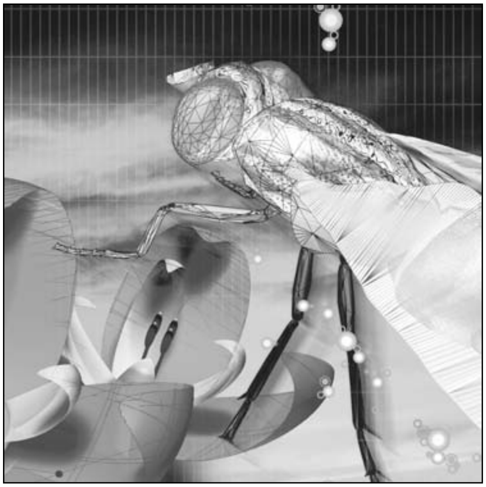
In this chapter, you've taken a look at some seriously professional ActionScript techniques. You've learned

- Best practices for modular, black-box programming using a few different techniques
- How to create modular movie clips
- How to use ActionScript to create real movement
- How to create a swarming effect and how to improve it using functions

More important, you finished the chapter by looking at how you can use modular code to break up the problems associated with a real and moderately complex website design, Futuremedia.

In the next chapter, you'll learn about sprites, and how to create and work with sprite functionality.





Chapter 10

GAMES AND SPRITES

What we'll cover in this chapter:

- Investigating what a sprite is
- Creating sprites, and assigning movement and collision behaviors
- Detecting a collision
- Creating the zapper game
- Animating the navigation of the Futuremedia site

This chapter covers sprites and ties them into the related building blocks that you learned about in the last two chapters. You'll look at basic sprite functionality and how to create it, before deconstructing an entire game so that you'll know enough about sprites to start making your own games.

In this chapter you'll create a Flash video game. This isn't the steep cliff-face at the top of the mountain it may seem at first; rather, it's the integration of the concepts you've examined in isolation in earlier chapters. You're going to take the separate areas of ActionScripting and put them together to create something that's much bigger than the sum of the parts. The glue that holds advanced ActionScripting together is your ability to see through problems with forward planning and your familiarity with the separate building blocks.

What is a sprite?

A **sprite** is a screen-based object whose motion and actions are controlled by dedicated code that's integral within itself. This means that the main code can let the sprite get on with whatever it's doing in terms of movement, collision, appearance, and so on, secure in the knowledge that the sprite can look after itself.

The sprite can be a space invader or a happy little spider that follows you around a website, helping you when you run into trouble. The sprite should be able to do three things without you having to keep an eye on it:

- Move
- Detect collision
- Act independently

The first point is that a sprite should be able to move around the screen in an intelligent way. It should know if it's moving into areas where it shouldn't be (such as off the visible screen area), and it should take its own remedial action.

The second point involves collision detection. A sprite should be able to detect when it has hit another sprite and take the appropriate action—for example, move away, explode, or cause what it hit to explode.

The third point involves the programming techniques you learned earlier when we talked about modular programming. The sprite should keep the main program informed of what it's doing and what its status is. This strongly implies that sprite routines should be designed using a **black box** or **modular** approach. The main program shouldn't have to bother with taking care of the sprite's movement or collision detection; it needs to know only what's necessary. The sprite is really just a self-contained movie clip that has methods that it applies to itself to control itself.

You've already covered the principles of something that goes beyond animation and moves into the realm of **simulation**. When you looked at inertial motion, you were simulating real motion, complete with acceleration, but that's only the start.

Simulation is the cornerstone for all that's related to advanced animation and specifically the way sprites interact with each other. To create a believable game world, you define rules for it, and everything in the game world has to conform to those rules. The interaction of the game sprites with each other is the difference between creating real-time games and creating separate advanced animated effects for websites.

For example, a ball that hits a wall has to know that it has hit a wall, and then it has to respond to this event. Similarly, a ball sprite that has hit a wall has to know that a collision has happened, and then it has to run code to react to this (i.e., it has to bounce away).

You've learned the principles that underpin the basic elements of sprite construction—it's just that you never knew you were a games programmer.

There are three basic factors you need to consider for sprite construction:

- Control
- Movement
- Collisions

Let's take them one at a time.

Control

Control is all about simulating intelligence and giving the impression that the sprite is aware of what's going on around it. Control is just a way of changing the properties of a movie clip (or anything else) so that the movie clip seems to act in an appropriate way. For the game you create in this chapter, you'll steal the swarm effect we looked at in Chapter 9 and reuse it for a swarm of aliens.

Now that you're giving the sprite some independence, you need to remember that it has to be aware of its own surroundings and their rules. It's not talking to the main program anymore, so it needs to know this itself. It's moved out into the real world now, and it has to do its own laundry. As well as knowing about what's going on around it, the sprite needs to be aware of its own internal status through its local scope. There are two separate *levels* of influence: external and internal.

External and internal data

Think of sprites as people. As people, we have internal influences, such as our own thoughts and emotions. We also have information coming to us to say what's going on in the immediate environment or television news about what's going on in another country. Our internal influences are the most important to us, but we need to be aware that we may have to adapt them in reaction to what's going on around us.

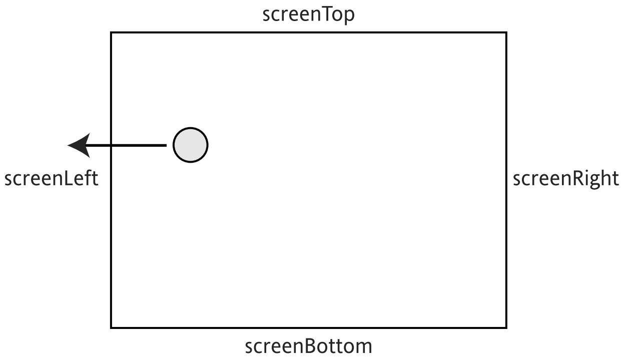
Similarly, a sprite's internally generated processes control it in the way our thoughts control us. At the same time, though, the sprite must also check whether its actions are beginning to break the external rules by looking at the immediate environment. The sprite doesn't have the same level of complexity we do. Its repertoire of interaction is as follows:

1. Set my next movement as per my controlling code.
2. Check whether I'm breaking any rules.
3. If I'm not breaking any rules, modify my external environment. Otherwise, move me as per step 1 (and keep repeating the three steps).

That might sound a bit vague, but consider your bouncing ball. It will keep moving forward, checking if it has hit something. If nothing is hit, the ball will move in its current direction. If something gets in the way, the ball can't continue forward. To do so would break a rule—the ball can't occupy the same space as something else. Instead, the ball will have to move away. It will have to *bounce*.

> *Note the subtle way the code version is different from real life. In real life, a ball is dumb. The ball bounces because there are physical laws that prevent it from going through anything it hits without changing direction. In the digital world, you don't have laws like that of the physical world, and impossible things can happen unless you force more sensible things to happen. The problem is that for a ball to be aware of physical laws and how they affect it, it has to know about them. It's no longer a dumb and passive ball, even if it looks this way on the screen. The digital ball has to know what the rules are before it can follow them, whereas a real ball simply follows them.*

Yeah, it's been a long time without a friendly sketch to help you! Let's fix that by looking at the steps graphically. The simplest way to illustrate these steps is to look at **boundary violation**.



The ball sprite in the diagram has a set of rules driven by code that causes it to move off the visible screen boundary. How do you tell it not to do that? One option would be to put some information about the size of the screen into the sprite itself, but that's not true to life. In real life, you know when you're about to hit a wall when you see the wall, rather than because you know that the coordinates of a wall are (234309, 324322) and you are at (234308, 324323) and about to hit it! Putting the size of the screen in the sprite also makes its responses less adaptable. In the future, you might want the sprite to change depending on a different external factor such as a differently positioned wall. Having to build everything again doesn't really fit in with the modular lessons that you've been learning.

In real life, you don't carry around information about your environment with you; you perceive the *environment itself*. An easier and more consistent way to tell the sprite not to move off the visible screen is to realize that the screen boundary positions are a **worldwide** or **external** set of attributes, or something that everything in the sprite's world should know about.

The contrast to this is **local** information that's contained within the sprite's immediate environment and not in the wider world. As an example, once the sprite moves, it assesses where its environment ends by looking outside of itself (or the `_parent` timeline). The sprite checks variables on this external timeline that tell it that it's violating an external rule, and it would modify its motion. In the same way, the internal thoughts of the sprite ("*I'm getting rather close to this other sprite*") are only required by the sprite itself, and if anything else wants to know this information, it should have to ask the sprite. You can see that information in your game world is also in two *levels*. The external information about the world is the lowest level, and as you get more and more specific, it gets less centralized and more concerned with individuals within the world. One alien sprite on one side of the screen doesn't really need to know that two alien sprites on the other side of the screen have just collided. It might put the former off trying to kill the evil human at the bottom of the screen!

The key word here is "levels," because it's something that Flash has already and something that you've been learning about throughout this book. You've seen that `_root` is the highest level and that the individual movie clips have their own timeline on which you can place (scope) information as variables. So, to know external information about the "world," a movie clip needs to look at the timeline outside itself, which could be `_root` but is more generally `_parent`. If the sprite needs to talk to or know anything from another inhabitant (sprite) within the world, it looks at the movie clip timeline owned by that inhabitant's movie clip. The world data is therefore structured around levels, and these levels are based on timelines.

There are actually three definite levels that a sprite can access:

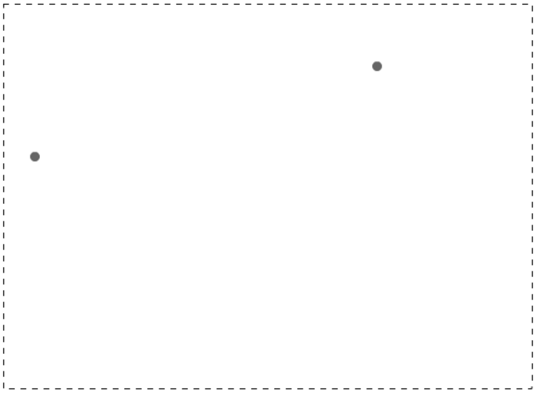
- `_global`
- `_root`
- `this` (the timeline of the sprite itself)

So what do you use each for? Global constants (such as the screen boundaries or physical constants such as gravity) will live in `_global`, with the shared data (which is defined as global data that's also dynamic or open to change) living in the main timeline the simulation is running on (which is invariably `_root`). Finally, local data lives on the local timelines, or `this`.

It's important when reading the scripts in this chapter to recognize which level each piece of information is on. The way to do this is as follows:

- Global data is preceded by g. This isn't something that Flash requires; it's a naming strategy we'll use to avoid confusion.
- Shared data has no path (and will be therefore placed on `_root`, because it's where the scripts are).
- Local data is preceded by `this` (which means that it will be placed on the timeline scoped by the current function, and this will usually be the timeline of the sprite itself).

If you want to see this in action, check out boundary.fla. It consists of two balls, one called slowBall_mc and the other called fastBall_mc. They live in a world that consists of the Flash stage area, denoted by a dotted set of lines.



The rules of this world are very simple:

- The boundaries of the world are defined by a set of global variables: gTOP, gBOTTOM, gLEFT, and gRIGHT.
- The two balls both have two variables on their timeline, xSpeed and ySpeed, and these variables define the speed at which the balls travel.
- The two balls use an external function, mover(), that controls their motion. This function looks at the timeline of each clip to see the value of the current ball's speed, and it also checks for collisions between each ball and the four boundary values, making the ball bounce back into the stage if a collision occurs.

The code for this world looks like this:

```
function mover():Void {
  this._x += this.xSpeed;
  this._y += this.ySpeed;
  if (this._x < gLEFT) {
    this._x = gLEFT;
    this.xSpeed = -this.xSpeed;
  }
  if (this._x > gRIGHT) {
    this._x = gRIGHT;
    this.xSpeed = -this.xSpeed;
  }
  if (this._y < gTOP) {
    this._y = gTOP;
    this.ySpeed = -this.ySpeed;
  }
```

```
    if (this._y > gBOTTOM) {
      this._y = gBOTTOM;
      this.ySpeed = -this.ySpeed;
    }
  };
  // set up global constants (globally available)
  _global.gLEFT = 5;
  _global.gRIGHT = 545;
  _global.gTOP = 5;
  _global.gBOTTOM = 395;
  // set up local variables (inside slowBall_mc)
  slowBall_mc.xSpeed = 4;
  slowBall_mc.ySpeed = 4;
  slowBall_mc.onEnterFrame = mover;
  // set up local variables (inside fastBall_mc)
  fastBall_mc.xSpeed = 12;
  fastBall_mc.ySpeed = 12;
  fastBall_mc.onEnterFrame = mover;
```

As you'll see when you run the FLA, the two balls move at different speeds. This is because they use their own *individual speeds*, as stored within themselves and denoted in the code by the two ball instance names. For example, `fastBall_mc.xSpeed` is a variable local to `fastBall_mc`.

The two balls live in the same world, however, and to simulate this they check against the *same* boundary limits. So you have two balls that

- Have different speed values, because this is localized
- Move around in the same world area, because they check the same world limits via the global variables
- Move in the same general way, because they pick up the same external movement function

Notice the way this hierarchy (which you've known about all along—it's just timelines, after all!) is becoming very powerful here. By changing the global values, you can *change what both balls do*. You can try this by changing the line

```
  _global.gBOTTOM = 395;
```

to

```
  _global.gBOTTOM = 295;
```

By changing the local variables per ball, you can change *what an individual ball does*. Change

```
  fastBall_mc.xSpeed = 12;
  fastBall_mc.ySpeed = 12;
```

to

```
  fastBall_mc.xSpeed = 24;
  fastBall_mc.ySpeed = 24;
```

and you'll see `fastBall_mc` get faster, but `slowBall_mc` is unaffected by the change. This hierarchy of black boxes within black boxes (or, in practical terms, movie clips within movie clips) is what makes advanced motion graphics in Flash tick.

The next issue surrounding sprite construction is a vital one: movement.

Movement

Movement is the most important attribute to the gamer, but to the programmer it's merely a consequence of control. You've already seen this in things such as the swarm code, which wasn't a movement so much as a section of code that controlled properties of a number of movie clips. The fact that these properties were actually associated with position is how the movement occurred, but you could just as easily have applied it to size, color, and so on.

We don't need to say much more about movement. Movement is a visual property that we think is important in real-time applications, but to us as programmers, it's really just a consequence of all the control and relationships that we've set up in our simulated world. Talking about movement in isolation isn't meaningful. Instead, you'll concentrate on the control you define that creates this movement in the first place.

If you want to create movement, you have to ask this question: "What's the code that's causing that movement, and what are the important features that I have to simulate to reproduce it?" You saw how this was done in the last chapter with the swarm movement, breaking down the swarming by deconstructing its movement back to a set of rules to do with (x, y) properties that are followed to create the movement in the first place. Now on to the last point: collision.

Collision

Collision is a vital ingredient in a believable game world because most sprites have to know something is near them before they can interact or change their responses in some way. In the game you'll look at, collision means just that. In other, more sedate games, it may mean something more fundamental: the initiation of complex interaction between two sprites. In other words, sprites don't necessarily have to try to blow each other up!

In the ecology game mentioned at the end of Chapter 9, collision would mean that two creatures in significant proximity start interacting. The two sprites would have to be able to signal to each other and retrieve information about exactly what type of creature the other one is. This information would allow the sprite to decide whether the other sprite was a potential mate, food, a predator, or just another like-minded creature with which to swarm for safety.

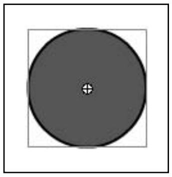
Collision is something new, so we'll show you in practical terms how it works in Flash. The file for the next exercise, `collision.fla`, is in this chapter's download file.

What is collision? Two things are in collision if their boundaries intersect. This is almost the same as a real-life collision. In real life, two things are in collision if their boundaries *touch*. Most collisions don't result in intersection because two solid things can't occupy the same space.

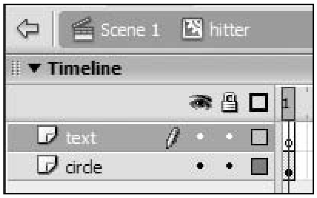
In the next exercise you'll create some basic collision detection. You'll make two circles and set up some dynamic text to tell you whether or not Flash sees a collision between them.

Creating the graphics

First you need to create the circle graphics.

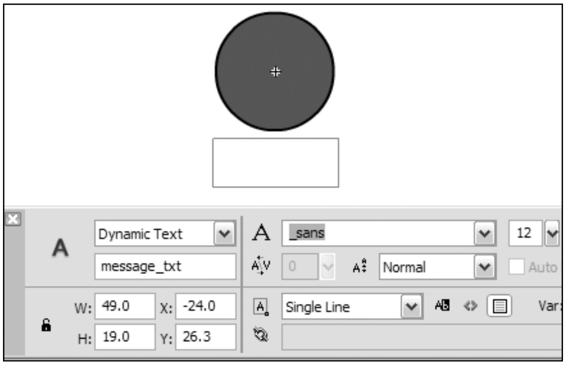


1. Create a new FLA using the default settings (550 × 400, 12 fps, white background).
2. Now you'll create one of the two movie clips that will be colliding. Draw a simple filled circle on the stage. Press F8 and make it a movie clip called hitArea. Give it the instance name hitArea_mc.
3. Next, you'll create the "hitter," the movie clip that will collide with the hit area. Draw another circle of about the same size as the first, and make it into a movie clip called hitter. Double-click this new movie clip to edit it in place. In the timeline of this new movie clip, rename the current layer as circle and create a new layer above it called text.



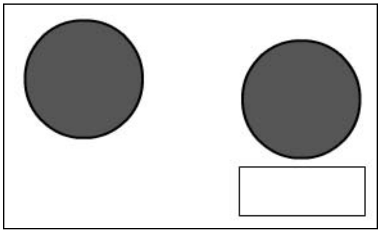
4. On the text layer, put a dynamic text field just underneath the circle. This will contain a bit of dynamic text set up to display a message that will tell you if Flash thinks a collision has taken place.

   With the dynamic text field in focus, select the Show Border Around Text button on the Property inspector so that the text has a border you can see. Deselect the Selectable button, and set it to display a single line. Give the text field the instance name message_txt.



Dynamic Text
message_txt
_sans
12
0
Normal
Auto
W: 49.0
X: -24.0
H: 19.0
Y: 26.3
Single Line
Var:

5. Go back to the main timeline and give the instance of the hitter movie clip the instance name circle01_mc.



Let's recap. You now have two movie clips on the stage. The circle-only one has the instance name hitArea_mc and the circle-with-text one has the instance name circle01_mc.

At the moment, the movie clips are some distance apart. You need to make one move toward the other somehow to cause the collision. The way you will do this is to make circle01_mc draggable, so you can drag and drop to move it.

Adding the code

Now follow these steps to add the code.

1. Back in the main timeline, create a new layer and call it actions. Rename the original layer as graphics. You now need to add some ActionScript to frame 1 of the actions layer. Enter the following code into the Actions panel:

```
function dragStarter():Void {
  this.startDrag();
}
function dragStopper():Void {
  this.stopDrag();
}
function detectCollision():Void {
  this.message_txt.text = "";
  if (this.hitTest(this._parent.hitArea_mc)) {
    this.message_txt.text = "hit";
  }
}
function attachHitterScripts():Void {
  this.onPress = dragStarter;
  this.onRelease = dragStopper;
  this.onReleaseOutside = dragStopper;
  this.onEnterFrame = detectCollision;
}
attachHitterScripts.apply(circle01_mc);
```

The code works by applying the functions listed in hitter to circle01_mc. The function dragStarter() becomes the onPress function, making hitter draggable when you click-hold over circle01_mc. The function dragStopper() makes it possible to stop dragging circle01_mc by releasing the mouse button.

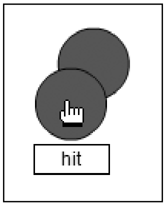
Note that the dragStopper() *function is assigned to both the* onRelease *and* onReleaseOutside *event handlers. This ensures that any mouse button release (with the mouse pointer either over the dragged clip or not over the dragged clip) will cause the clip to be dropped. Specifying both* onRelease *and* onReleaseOutside *is generally very good practice in your Flash projects.*

The detectCollision() function is where the interesting code is. This function is used as the onEnterFrame of circle01_mc, and it will place the text hit in circle01_mc's text field if a collision is detected. If no collision is detected, the text is made to show nothing. The actual detection is made via this line:

```
if (this.hitTest(this._parent.hitArea_mc)) {
```

The hitTest method will return true if "this" (i.e., circle01_mc) hits the hitArea_mc movie clip, noting that hitArea_mc is on the parent timeline to "this" (i.e., _root). The method returns false if no collision is detected, so in this case, the text remains blank.

You can try the code to see this. Drag the circle01_mc movie clip over hitArea_mc.



Some of you may be wondering why we used named functions and this:

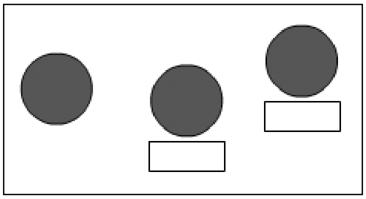
```
attachHitterScripts.apply(circle01_mc);
```

instead of this:

```
circle01_mc.onPress = function() {
  this.startDrag();
};
circle01_mc.onRelease = function() {
  this.stopDrag();
};
circle01_mc.onReleaseOutside = function() {
  this.stopDrag();
};
```

and so on. The answer is coming up.

2. Add a new instance of the hitter movie clip called circle02_mc. You now have three movie clips on the stage.



3. Give the new instance the instance name circle02_mc. At the end of the script so far, add the following:

```
attachHitterScripts.apply(circle02_mc);
```

If you test the FLA now, you'll see that both the existing and new hitter instances work. You've found a way to attach a number of events to any new movie clip with just one line!

What you've actually done is made your code more general. The functions that become event handlers don't care about the name of the movie clip—they use "this." The function attachHitterScripts() *also doesn't care for the same reason, but it does know which functions have to be assigned to which event to make the whole thing work. Finally, the* apply() *method makes the* attachHitterScripts() *function run with the scope (i.e., which movie clip will be substituted for "this") equal to the argument—in this case,* circle01_mc *and* circle02_mc.

Also worth remembering is that apply() *isn't a method of the movie clip; it's a method of the* Function class. *Yes,* Function *is also a class, just like* MovieClip. *That may seem a little odd, but if you think about it, it makes sense. Functions are created and are attached to timelines just like movie clips. They have instance names (*attachHitterScripts, dragStarter, *and so on). What they don't have is symbol names, but that's because they aren't symbols—they're purely code, and they don't need to live in the Library. This is all part of the general rule that almost everything that ActionScript knows about is a class. ActionScript doesn't understand much else apart from instances and classes.*

Remember, if at any time you get lost or just want to check that you're doing things correctly, open collision.fla *to check your work against the original file.*

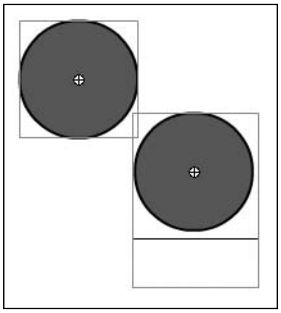
This would make a nice drag-and-drop interface. You could have the user physically drag and drop products (which would be the equivalent of the circles) into a shopping basket, for example, or have an MP3 downloads page where the user chooses music through record icons he or she drops into a DJ's record bag.

There are some issues to note here:

- You have to name the item you want to detect the collision with, so you must know its instance name. This means that you can't just detect a collision with "anything" as the target (like you can in real life—if you walk into an obstacle, you'll hit it). To do this in ActionScript would require a loop that looked at all available instances, which would mean that you have to know the names of all available instances at any time.
- The two hitter movie clips don't detect collisions between each other; rather, they detect collisions between themselves and hitArea_mc.
- You use a general set of functions that control the individual parts of the problem, but individually these functions (dragStarter(), dragStopper(), and detectCollision()) don't do anything to solve the problem. This is left to the function attachHitterScripts(), which knows which function to attach to which event for each movie clip it's applied to. The thing that tells it which movie clip is the *scope* that it's running in. The scope is defined by attachHitterScripts.apply(). This calling action is used by the attachHitterScripts() function via the path this. Thus you have a hierarchy of code, starting with apply(), then attachHitterScripts(), then the three functions that actually do the work.

- This exercise provided a good example of structured coding and included something new: black boxes within black boxes. At the apply() stage, you simply call attachHitterScripts(), and you *don't need to know what the scripts* attachHitterScripts() *will call are*. Inside attachHitterScripts(), you see the names of the three scripts, because you've moved down a level into a box that needs to know the names.

You'll notice that the hit detection sometimes returns true when the circles are close but not touching. This is because you're checking the **bounding boxes**, which are the little outlines that appear around symbols when you select them in the Flash authoring environment.



You can modify the hitTest method to check for movie clip outlines rather than the bounding box, but this is difficult and makes for a more complex collision-detection routine. Let's get it working with bounding box collisions for now.

There is one final area to consider in interactivity: performance. At the moment, the circles are updated at 12 fps, and this is a little slow. There are two ways around this. One is to increase the fps to something higher, but this makes Flash do *everything* faster, and it's a little inefficient. Good programming doesn't rely on making things better by using speed (because you'll soon find out that you can never have enough), but on making code *more efficient*.

> *Increasing the frame rate to silly values because "it seems to run better" is a common technique. It's OK for small animated effects, but if you're building serious Flash applications, you need to accelerate only some of the things in the movie. Accelerating them all makes Flash Player work harder and leads to posts on Flash message boards with questions like "Why is Flash so slow?" The answer is, of course, that the developer is expecting Flash to cover over inefficient code by running just that bit faster if the fps is set higher. If only everything was this easy!*

The Flash action updateAfterEvent() will add an extra screen redraw whenever it's invoked. You can use this action only inside (and at the end of) an event script, as the name of the action itself suggests. That kind of sounds useful, but where do you want to use it?

What you actually want to do is add extra frames only when the user is dragging a circle. You know when that occurs and for how long, because

- When the circle in question has been clicked *and* the mouse is currently moving, you know that the user is currently dragging.
- When the circle in question has been dropped, nothing is being dragged, so you can stick with 12 fps.

So once the mouse is clicked over a circle, you want to start looking for mouse movement and use updateAfterEvent() every time you detect this. In ActionScript, this converts to create an onMouseMove event that updates the screen when the mouse moves.

If the mouse button is released, then the user has stopped dragging. You no longer need to force Flash to use updateAfterEvent(), and the easiest way to do that is to delete the onMouseMove. Have a look at collision2.fla for an example of this in action. The updated code function is shown here, with new lines in bold:

```
function dragStarter():Void {
  this.startDrag();
  this.onMouseMove = updateAfterEvent;
}
function dragStopper():Void {
  this.stopDrag();
  delete this.onMouseMove;
}
function detectCollision():Void {
  this.message_txt.text = "";
  if (this.hitTest(this._parent.hitArea_mc)) {
    this.message_txt.text = "hit";
  }
};
function attachHitterScripts():Void {
  this.onPress = dragStarter;
  this.onRelease = dragStopper;
  this.onReleaseOutside = dragStopper;
  this.onEnterFrame = detectCollision;
};
attachHitterScripts.apply(circle01_mc);
attachHitterScripts.apply(circle02_mc);
```

The way the dragStarter() *function handles the issue of jerky drag-and-drop animation is to add extra frames only when the mouse is moving. This means that the "hit" text fields are still being updated at 12 fps, but the circle animation is animated much faster (it's animated as fast as Flash can run). Targeting only the stuff that you actually want to run faster has a cool side effect: because Flash doesn't update everything as fast as it can, the "as fast as Flash can" ends up being much higher!*

If you run this FLA, you'll see a marked improvement in the animation speed.

You've had a look at fairly simple examples thus far. Now it's time to delve into a bigger example, something more like the sort of project you'll want to create yourself.

Planning zapper

In this section you'll create a space invaders–type game, with swarming aliens at the top and the player's ship at the bottom. This is a pretty well-known setup, so we won't waste your time by showing you a design breakdown of a game that's been in the public domain since the late 1970s.

Just because we're not displaying a design breakdown doesn't mean we haven't thought it through. To shape the initial concept and develop an idea of what would happen, we thought of all the arena games we've played and broke down all the functions. As stated in Chapter 2:

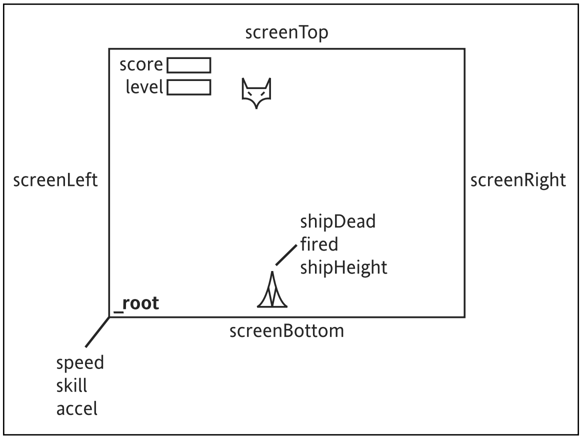
> *If it doesn't work on paper, it won't work when you code it.*

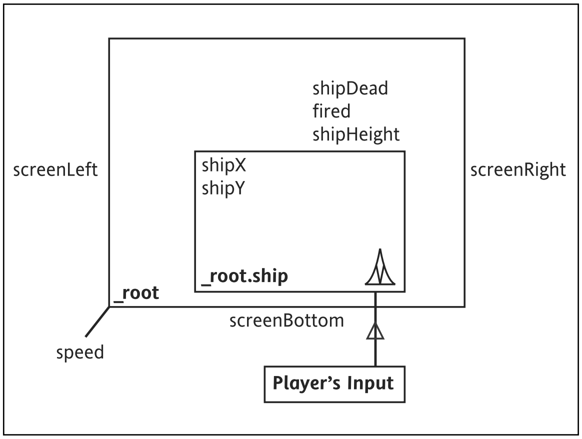
For simplicity's sake, the player will be allowed only one life and one bullet (so that there's only ever one bullet on the screen). Also, the invaders won't attack by firing on the player. This is entirely possible, but we wanted to avoid defining a game with too many sprites on the screen at the same time. Instead, the aliens will perform some form of swooping or diving attack, which calls for fewer sprites and makes for a faster game.

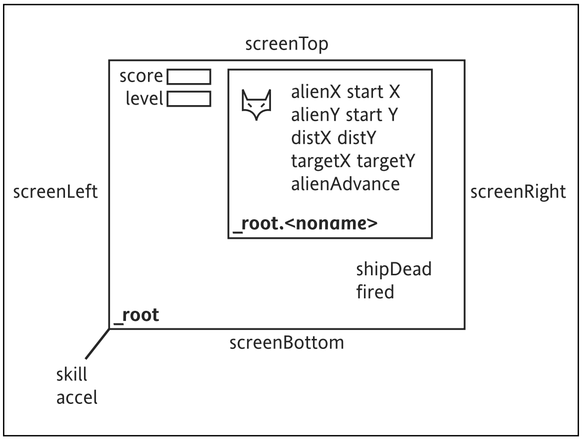
One of the major reasons that Flash is at a disadvantage when it comes to games is that it's forced to play its game inside a browser window and it can't control the screen attributes, such as what size the full screen will be. Unlike dedicated consoles that play games on fairly small screens, usually no bigger than about 600 × 400 pixels, Flash has to play on a screen with an unknown screen resolution. This could be a screen size of 1280 × 1024, on which currently only the highest specified desktop computers can render fast games acceptably, and then only with some form of graphic hardware support. Flash has to do this on the browser, a platform that wasn't built for speed. The Flash player itself doesn't do you any favors, as it doesn't grab all the available processor time for itself as most video games do. So, you'll keep things simple for the moment.

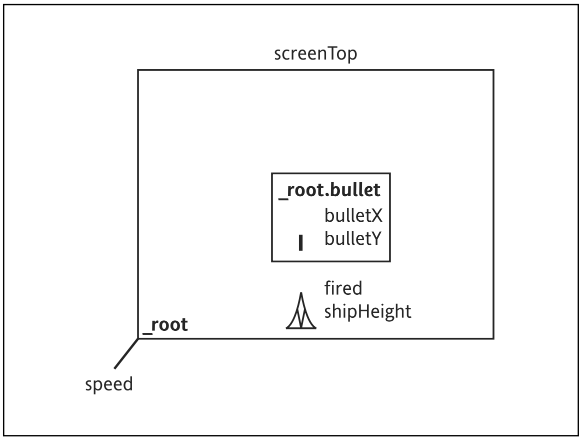
In the early arcade game machines, the games lasted a calculated amount of time for the average player. The difficulty level was graded so that the player would think she had gotten her money's worth. An important consideration at the planning stage—and one often overlooked by Flash games—is how long you want an average game to last. This affects all sorts of usability questions such as "Is the game too challenging?" "Does the game last long enough?" and "Does the player see enough of the game on the first turn to want to play again?" A good general assumption is that a good web game should last no longer than one minute for the first time, have immediately obvious controls, and have no long preloads that break up the game. The player should also be able to get better from her abysmal start by practicing.

The next step is to plan how the game will work from a programming perspective. The block diagrams that you see here relate to different aspects of the game and are taken directly from the three pages of handwritten notes and sketches created before writing the game.









You'll come back to look at each of these diagrams in more detail as you examine the ActionScript behind each section. We wanted to show them to you here so that you could see how they're as useful as the flowcharts that you looked at in Chapter 2, but on a slightly higher level. Flowcharts are more concerned with detailed logic, whereas these block diagrams are high-level overviews that help mesh together a game world with no logical flaws. Once we had these diagrams, we dropped down a level and began flowcharting the smaller ActionScript pieces of that world.

These diagrams are a lot more useful than flowcharts because they imply *relationships* rather than logic, and they're easier to formulate. They show objects and their interfaces, which is useful for getting an overview of what each game piece can see and interact with in the game world, and what in the game world will influence each game piece's behavior.

It took about half an hour to create these diagrams, and by the time they were finished, all the information needed to complete the game was right there. This is the way you'll need to work if you want to reach this level of ActionScripting, and it's more or less mandatory if you expect to do it commercially. It took about eight hours to write this game, including planning, which isn't far from the timescale that you'll be used to in the real world. To turn ideas around that quickly, you must plan everything out first in the way shown here. At first it may seem to slow you down, but with time you'll be able to home in on the important bits very quickly, and you'll find that the time you spend planning reaps massive savings when it's time to code.

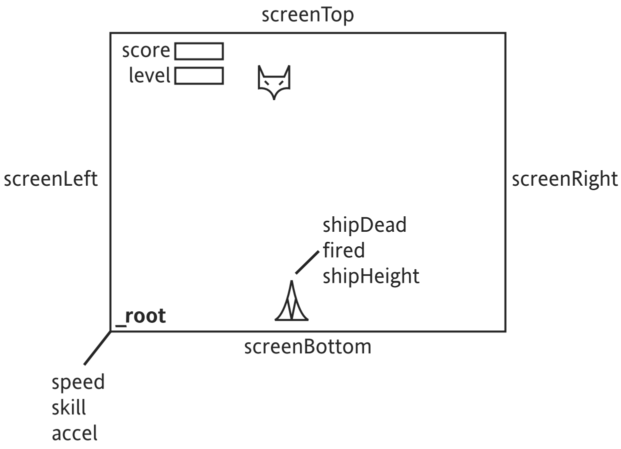
We don't propose to go through the whole program as a tutorial in this section, because you should have developed far enough in ActionScript at this stage to be able to understand what's happening. We're guessing that you're more interested in seeing how the plans are converted to ActionScript so you can go away and do the same thing with *Missile Command* or *Defender*. We'll break down the code and tell you what each bit does and the reasoning at every stage.

As previously mentioned, you need to plan all aspects of a game before coding it. Here, we'll show you the plans and the code separately for each section so that you get a clear view of what the ActionScript is doing in each section. Don't think for a moment that we planned only one section at a time!

The game world (the main timeline)

One of the important issues in designing your game world is **approximation**. You want to concentrate only on those things that are important in your game. Deep space has all sorts of things that you aren't interested in, such as temperature, radiation intensity, and so on. These things won't affect anything in your game world, so you don't need them. This is an important thing to have a feel for, and it's one of the defining features of good simulation: knowing what to leave out and what to leave in.

The things your game will need to address are shown in the following image. You'll look at the player's ship, the aliens, the stage, and the score/level.



gSCREEN_TOP, gSCREEN_BOTTOM, gSCREEN_RIGHT, and gSCREEN_LEFT

These are the game world limits and represent the visible screen area. They're all constants, so in the final code they're shown in CAPITAL_LETTERS. We've also placed them in the `_global` level because they're universal constants for the game world (hence the lowercase g). You'll see later that the game allows some sprites to start off outside the game world, but their controlling code will force them to move back into the game world.

score

`score` is there for the player only and represents the carrot that keeps the player wanting to play again. As you'll see, `score` also feeds into the `level` parameter.

level, skill, and accel

`level` is an important game world parameter, because it controls how well the SwarmAliens will play. It's derived from the `score`, and the higher the score, the higher the level. The longer the player survives, the harder the game gets. `level` works by feeding into several other parameters that control the alien behavior: `skill` and `accel`(eration). As the aliens get more skillful, they'll attack more, and as the `accel` parameter changes, the aliens will get to their top speed faster and therefore get from point A to point B quicker.

speed

speed is how fast things move in pixels per frame. The aliens will have a slightly different movement strategy, based on the equation in the last chapter, but the stars and player's ship will have movement based on this parameter.

shipDead, fired, and gSHIP_HEIGHT

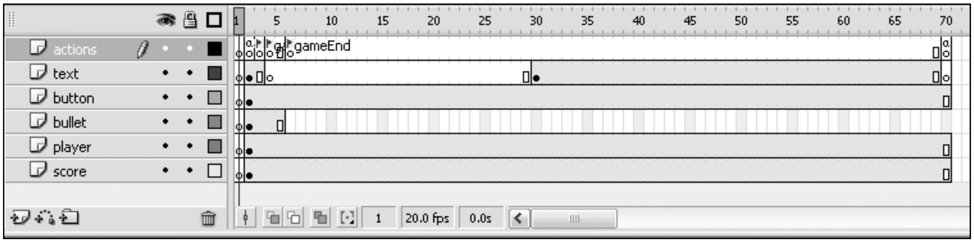
These are some parameters for the player's ship that will be needed by the other sprites, so we decided it would be more convenient to make them sit on the main timeline as external variables. First, `shipDead` represents whether the player's ship has died or is still fighting the good fight. `fired` tells you whether a bullet is in the process of being fired (i.e., the bullet is onscreen and visible).

We created `gSHIP_HEIGHT` as a global variable because we could see it being important for at least some of the other sprites. One of these would be the bullet, which needs to start off at the top of the ship because that's where it's fired from. We also thought that it might come in handy if the game runs too slowly—the aliens can't hit the ship until they're `gSHIP_HEIGHT` above `gSCREEN_BOTTOM`. You don't really need to test for collisions between the aliens and the ship until then. As it happened, the game ran fast enough and we didn't need to do this, but if you decide to develop the game further—and we hope you do—then you have this option to consider.

Let's have a look at how this game is coded.

The timeline

Open zapper.fla from the download for this chapter and have a look at the main timeline.



The first thing to notice is that the frame rate is set to 20 fps. This is a normal frame rate when smooth graphical movement is at a premium.

> *The reason frame 1 of the timeline is blank has to do with preloading. You want everything to be in the Library before the scripts start running; otherwise, the game may not work properly.*
>
> *Normally in online Flash content, the next frame on the timeline isn't played until all content required to display and animate that frame (movie clips, graphics, buttons, scripts, and so on) has been loaded into the browser. This loading occurs in the current frame. Frame 1 is special because it has no frame before it for this loading to take place, and for advanced Flash content (games and other ActionScript-heavy content), this can occasionally cause problems.*

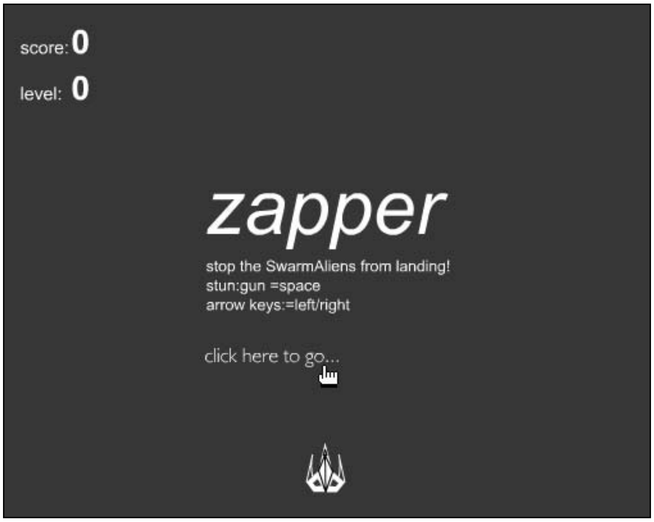
Most of the game plays out in the first six frames of the actions layer:

Almost all the code is on frame 2 (comment label gameInit) of layer actions. This frame initializes the game, setting up the functions and events that drive zapper.

> *A comment label is a keyframe with a label that starts with //. A comment label isn't exported into the SWF. Like comment lines in ActionScript, comment labels are there for your benefit.*

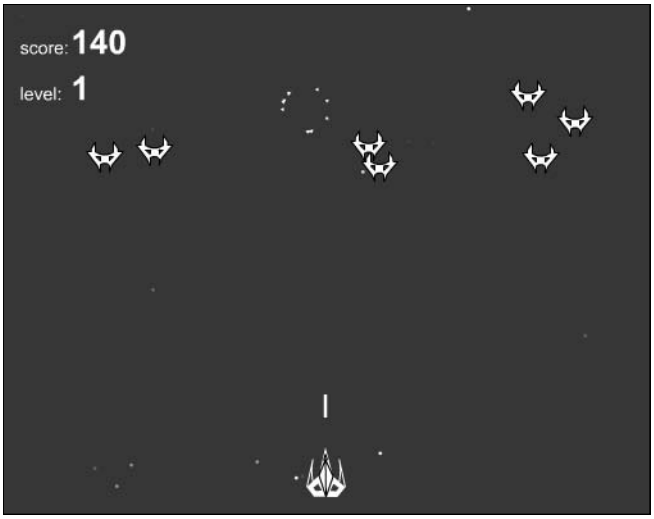
Frames 3 (label attractor), 4 (label gameStart), and 6 (label gameEnd) contain the three main phases of the running game.

attractor is the start screen. The game is paused at this frame, waiting for you to click the start button (an invisible button over the click here to go text) and begin a new game.

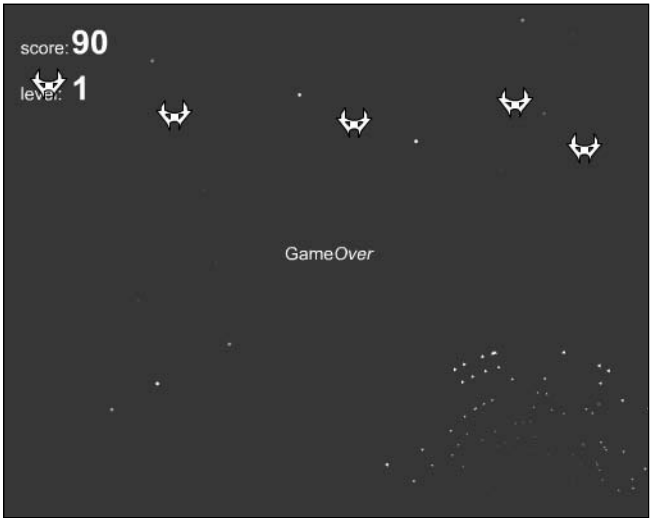


The introductory screens in coin-based arcade machines are called the ***attractor*** *because they're designed to entice folks to put their money in the slot and play the game.*

gameStart defines the start of the game. In terms of code, this frame initializes each new game. Some variables (such as the player's score) need to be reset on starting a new game.



gameEnd indicates the end of the game. This is signified by a variable (shipDead) being set to true, telling you that the player has been hit.



The code

The majority of the code is attached to frame 2 of layer actions. Select that frame to see the script in the Actions panel.

As usual, the main code starts at the bottom of the script, given that it's normal for functions to be defined first. Let's skip to the main code.

Global constants

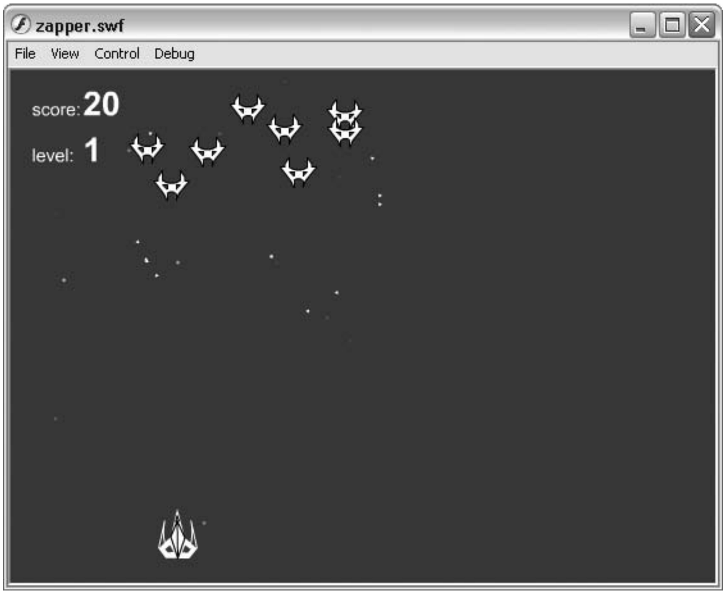
Scroll down to about line 215 to see the following code:

```
// set up screen extents
_global.gSCREEN_LEFT = 20;
_global.gSCREEN_RIGHT = 530;
_global.gSCREEN_TOP = 20;
_global.gSCREEN_BOTTOM = 380;
_global.gSHIP_HEIGHT = 40;
```

Here's a little tip. While focused on the Actions *panel, you can use Ctrl+G/Cmd+, to bring up the* Go to Line *dialog, which will allow you to enter a line number and skip straight to it. Neat, eh?*

This code sets up the global or "universal" values, and they are constant. They're universal in the sense that they're the things that define how the game world will work, much like physical constants define the real world. They're therefore defined in the `_global` scope. You can see how they affect the game by doing the following:

1. Test the SWF. You'll see the game start playing on the main stage. While you play the game, notice that all the sprites stay within the stage. In particular, they seem to know when they're near the right side of the stage, and they don't go outside it.
2. When you lose, close the SWF, select frame 2 of the actions layer and open the Actions panel. Change the value of the `gSCREEN_RIGHT` variable as shown here: `_global.gSCREEN_RIGHT = 300;`. Now test the movie again and you'll see the difference this has on the game:



You'll no doubt have noticed that *everything* in the game now sees a different right edge and reacts differently.

The variables you've chosen as universal all have this property of being able to change the way *everything* in the game works, and this is why they're in `_global`: everything important in the game uses them, so they need to be available everywhere.

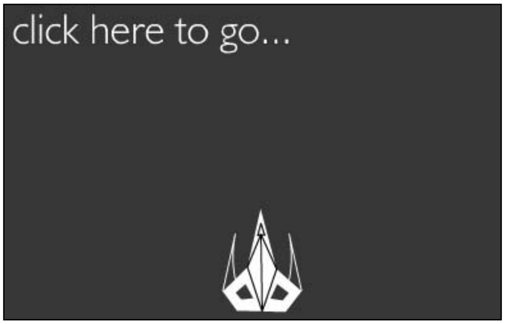
In most modern games, any particular title is built in two parts: the game engine and the game data. The game engine is general and can be used to build several different games or to emulate different environments in the same game. One of the ways this is achieved is the same way you changed a global variable just now to change how the game world reacts. For example, if your hero moves from planet earth to the low-gravity alien home world in the final level, you can modify the effects of gravity by changing one thing: the global constant that controls gravity.

3. Before we proceed, close the movie and change _global.gSCREEN_RIGHT back to 530 in the code.

The "start game" trigger

The main code in the game appears directly after the global constants. The first thing you do is set up the event handler for the button that starts the game. Notice that this isn't the simple gotoAndStop("home") type of event handler you may have used in the past. The event handler is a whole series of function calls, and these will result in *hundreds* of lines of ActionScript running.

```
// event handler for invisible button
invisible_btn.onRelease = function() {
  this.enabled = false;
  initGo();
  makeStar(15);
  makeAlien(10);
  bullets.apply(bullet);
  starShip.apply(ship);
  gotoAndStop("gameStart");
};
```



You'll look at the more complicated individual functions in turn a little later, but here briefly is what the button script does.

First, the button disables itself. Why? Well, once the game starts, you no longer need the button, but if you remove it from the timeline, you also remove the event handler shown. Rather than the beginner "move to a keyframe where it doesn't exist" way of removing content from a timeline, you're now using a more ActionScript-centered way of controlling things. You don't use timelines as much, but you change the thing by varying its properties. In this case, you simply disable the button.

```
this.enabled = false;
```

The button is invisible, but if it wasn't, you could make it invisible as well as disabled by adding the code this._visible = false;.

Next, you initialize for a new turn of the game by calling the function initGo(). This sets up the variables that need to be reset every time so the player miraculously lives again with a zeroed score (shipDead = false rather than true, score is zeroed), the aliens revert from the super-killers they became at the end of the last game to the pussycats they always seem to begin as (speed = 10, skill = 0.04, accel = 20), and so on. The function itself starts around line 200 and looks like this:

```
function initGo():Void {
  // reset variables for this turn
  speed = 10;
  skill = 0.04;
  accel = 20;
  score = 0;
  level = 1;
  fired = false;
  shipDead = false;
  // clear score and level text
  score_txt.text = score;
  level_txt.text = level;
}
```

So to recap, the start game button has been clicked, and so far you've run the code to set up your game data (variables). What you haven't yet done is create the graphics. You do this via the next two lines:

```
makeStar(15);
makeAlien(10);
```

This does exactly what it implies: it creates 15 stars (as can be seen scrolling in the background) and 10 aliens. You should play around with these values, adding higher values if your computer is fast enough to handle them. If all else fails, however, you could use good old bitmap caching to keep the frame rate of the movie up.

This is, of course, modularity at work again. Although you have no idea how the code to create stars or aliens works (or even, at this stage, where it is) you can use it via its black-box interfaces. All you need to know is how many and the code that sits in the makeAlien() *box will do it all for you! You'll look at the* makeStar() *and* makeAlien() *code in a moment, so don't worry, you'll peep inside the boxes eventually.*

So now you have variables, stars, and aliens. All you need to do is get your ship powered up and your lasers online to thwart the invasion. You do this via the last two lines in the button event handler:

```
bullets.apply(bullet);
starShip.apply(ship);
```

The first line attaches the function called bullets() to the movie clip called bullet, and the second line attaches a function called starShip() to the movie clip ship. By the magic of modular code, you don't have to know what the functions bullets() and starShip() do just yet, except that the first makes the bullet movie clip act like a bullet, and the second makes the ship act like a starship in a space invaders game. There—done!

Um, not quite. You've set up all the different *parts* of the game: stars, aliens, player, and bullets. What you haven't done is tie them all together so the game can be played. Rather like a baseball game where the players are dressed for the part but nobody knows how to *play* baseball, nothing much will really happen in your game—or rather it will, but not in a coherent way. For example, the aliens will dive at the player, but the game doesn't know what should happen when an alien actually hits the player.

What you need are some *rules*. You implement these rules in the zapper game via an onEnterFrame script that keeps track of what's happening via a check every frame.

```
// set up rules
this.onEnterFrame = gameRules;
```

The actual function called up as the umpire script is around line 185 onward and looks like this:

```
function gameRules():Void {
  // game rules...
  if (score > (level*200)) {
    skill = skill*2;
    accel = accel/1.5;
    level++;
    level_txt.text = level;
   }
   if (shipDead) {
    delete this.onEnterFrame;
    gotoAndPlay("gameEnd");
  }
}
```

There are two rules in the game:

- **As the game progresses, the aliens should get more aggressive**. The first if checks the current score and sets up variables that will make the aliens more aggressive based on it. The visual indication of this aggressiveness is the game level. At level 1, the aliens are placid, but they get impossible by level 5.
- **If the player's ship gets hit, it should explode and the game should end**. The second if checks the variable shipDead, which is set elsewhere if the ship is ever hit by an alien. If shipDead is true, you set up conditions for the game to end: make the main timeline play from label gameEnd and stop the umpire script (this.onEnterFrame) from running (it will start again the next time a game starts via the start button).

> onEnterFrame *is attached to this, where this is the timeline you're currently on,* _root. *As* _root *is the game world inside which all other movie clips exist (and inside which all your game-control variables exist), it seems reasonable that this should be the timeline that controls the game. The game world (*_root*) maintains itself!*

Finally, you set up the timelines so that the appropriate graphics show. The ship movie clip has a tween explode sequence on its timeline that you want to play when the ship is destroyed. You don't want to play this sequence just yet, so you stop its timeline. You also move the main timeline to the start of the game proper: the **attractor** frame.

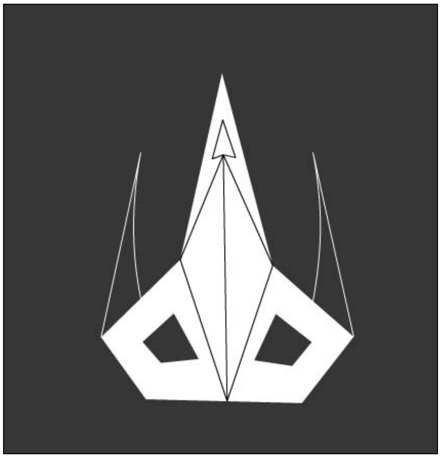
```
// move timelines...
ship.stop();
gotoAndStop("attractor");
```

Note that although the scripts are defined at frame 2, they actually start working on the attractor screen. The first frame on which the user can click the start button is frame 3.

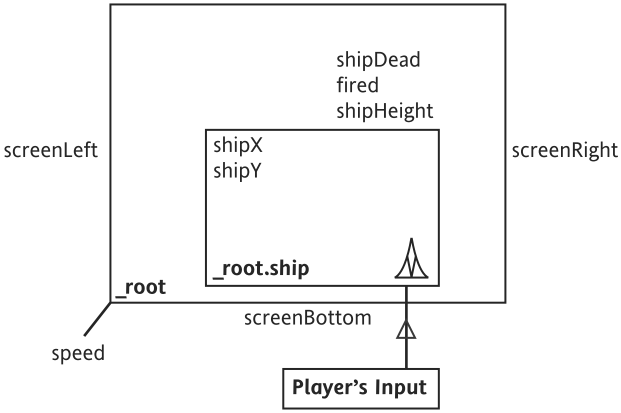
Now all the main code is sorted out. Next up are the individual sprites.

The player (the ship)

The player's sprite has no intelligence of its own; rather, it takes its movements from the player's keypresses. All it has to do is move left and right, and fire on demand from the player. It also has to check that it's always within the game world.



That's not enough information to actually write code, though. For that you need a diagram that tells you the same thing in terms of what ActionScript knows—that is, *data*, or in this case, variables.



This diagram defines your ship sprite in terms of its data. There are several things of note on this diagram:

- **The data that's used to control the ship**. All variables used are shown here. You can also see where they are (i.e., their scope) in the final SWF by where they are in this diagram. You know shipDead is on _root and shipX is in ship because they're in their appropriate movie clip "boxes." You also see that screenLeft is a global variable because it isn't in a box (i.e., it has no scope, which makes it _global and available everywhere).
- **Important external variables**. All variables outside ship that are particularly important are placed in the bottom-left corner, with a label showing where they come from. The variable speed is very important, and you can see that it's in _root.
- **The hierarchy**. You can see that the ship movie clip is inside _root because the ship box is inside the _root box.

From the diagram, you can see that the ship movie clip is controlled by its own variables, shipX and shipY, and that the variable speed (on _root) is also particularly important. Other variables used include shipDead, fired, shipHeight, screenRight, screenLeft, and screenBottom (rather, the CAPITAL_NAME versions in the code). You can also see that the ship control code uses inputs from the player. Best of all, you see this *graphically*.

*Although you'll probably learn about flowcharts in programming class, the graphical diagrams you're most likely to see in real industrial code are more complex versions of the diagram shown previously. This type of diagram goes by several names, but the one that best describes it is **object block diagram**. It shows the relationship between all the objects (instances) in your code. It's also very useful for debugging your code.*

*For example, if you find that your ship isn't firing properly, you could look on this diagram and quickly note that there's a `fired` variable. You could then look at all the other diagrams to see where `fired` is coming from (sourcing) and where else it's going (sinking). By looking at what the other sink instances are doing, you can capture a lot of information about the error behind this simply by looking at the object block diagram, **before** you start looking at any code. Examining high-level object block diagrams and working down graphically toward the actual code is called **system engineering** in some fields. It involves treating code (and most other things) as blocks and working out what's going on by looking at the interfaces between the blocks.*

The ship code is split into two different sets of code (both of which are within the main function `starShip()`): the `onEnterFrame` handler and some initialization code. Then you apply `starShip()` to the `ship` movie clip, and you can assume that the code contained in `starShip()` is on the `ship` timeline (even though the code is actually on `_root`; scope is a wonderful thing in Flash when you master it!). Thus, the function `starShip()` is not so much a function in its own right, but more like a code block that will eventually be scoped to the `ship` timeline.

Put a different way, it's easier to follow the script if you assume all the code inside the function `starShip()` is attached to frame 1 of the `ship` movie clip timeline. It isn't in the FLA, but scope will make it act as if it is in the SWF when you run the game.

This is a very powerful concept. It allows you to write code that will relocate to where it's needed in the final SWF, even though all the code is in one manageable and easy-to-find script in the original FLA.

The main code for the `ship` instance is at the bottom of the `starShip` function (around line 90 onward):

```
// Initialize ship on first run...
this.leftArrow = 37;
this.rightArrow = 39;
this.space = 32;
this.shipX = (gSCREEN_RIGHT-gSCREEN_LEFT)/2;
this.shipY = gSCREEN_BOTTOM;
this._x = this.shipX;
this._y = this.shipY;
this.gotoAndStop(1);
```

leftArrow and rightArrow are key codes that will be used later to detect presses of the spacebar, the left arrow key, and the right arrow key on the keyboard. They're used to get the user inputs coming from the Player's Input block in the object block diagram.

> *You can see a list of all the key code values in the* Help *panel. Search on the phrase "keyboard keys and key code values".*

The variables shipX and shipY contain the (x, y) position of the ship. They're set to initial values by looking at the world size constants. To actually move the movie clip to this position, you set the _x and _y properties of the ship movie clip at frame 1.

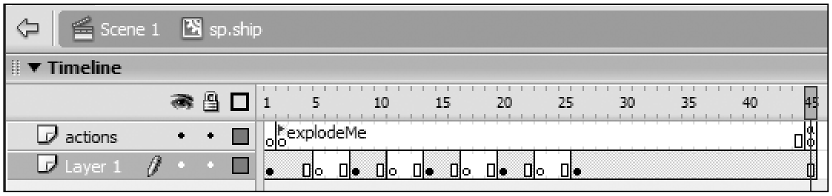
The main code controlling the ship is contained within the onEnterFrame definition starting at about frame 70:

```
this.onEnterFrame = function() {
  if (Key.isDown(this.leftArrow) && (this.shipX > gSCREEN_LEFT)) {
    this.shipX -= speed;
  }
  if (Key.isDown(this.rightArrow) && (this.shipX < gSCREEN_RIGHT)) {
    this.shipX += speed;
  }
  if (Key.isDown(this.space) && (!fired)) {
    fired = true;
    bullet.bulletX = bullet._x = this.shipX;
    bullet.bulletY = bullet._y = this.shipY-gSHIP_HEIGHT;
  }
  if (shipDead) {
    this.gotoAndPlay("explodeMe");
    delete this.onEnterFrame;
  }
  // ship movement
  this._x = this.shipX;
};
```

Although the code looks a little long, it's just four conditions plus a line at the end to actually animate the ship. In plain English, the code is saying this:

```
If the player is pressing the left arrow key and I am not too far ➥
 left already, go left.
If the player is pressing the right arrow key and I am not too far ➥
 right already, go right.
If the player is pressing the spacebar and I am not firing ➥
 already, fire.
If I am dead, start my dying animation and kill my controlling script.
```

If you look in the ship's timeline (`sp.ship` in the timeline, with `sp` being our chosen identifier for "sprite"), you'll see that it contains three keyframes in its actions layer:

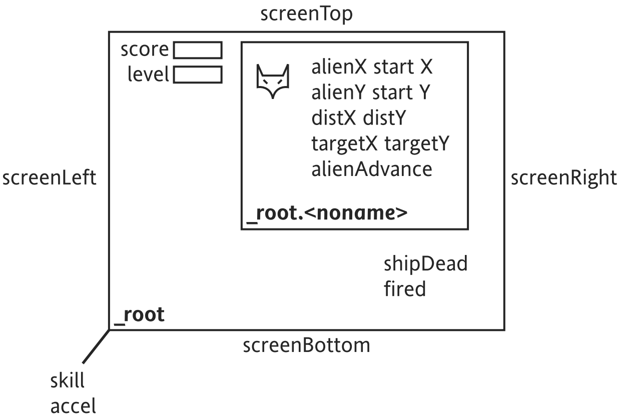


The timeline will stay on frame 1 (the "ship is OK" graphic) until the controlling scripts make it play. When this happens, the ship's timeline will play from frame 2, resulting in the "blink and explode" animation, then it will stop at the last frame and wait until the ship is made to restart at frame 1 of its timeline, and do the whole thing again.

The SwarmAlien

In many ways, the SwarmAlien is the only thing in the game with intelligence. Its movements are based around a real motion that uses acceleration, and it defines its own motion (as opposed to the spaceship, which is really no more than a puppet controlled by the player), attacking or swarming based on a fairly advanced script.

The SwarmAlien also advances down the game world as time goes on, getting closer and closer to the hapless player. It's only a matter of time . . .



The SwarmAlien makes all the collision detections in the game, which is fairly impressive considering it's the main target. Altogether, it's quite an intelligent life form!

In logical terms, in this game the aliens are hitting the bullets rather than the other way around. It's easier to do it this way because each alien has to make one check for one bullet, rather than the single bullet having to check for all aliens. This seems a bizarre way to model the bullet-hitting detection, but it actually results in less code.

The SwarmAlien uses the four global constants that define the stage extents to make sure it stays onscreen as much as it can (it starts offscreen, so its first action is to fly into the visible stage area as soon as it can). It has a number of internal variables (alienX, alienY, startX, startY, distX, distY, targetX, targetY, and alienAdvance), although you've seen most of them before in another form. They're very much like the swarm code you saw earlier; the SwarmAlien is based around the inertia/flocking motion. The alien has no name because both the alien and its name are generated dynamically at runtime.

In terms of external variables, the alien is highly dependent on the variables skill and accel (which define how fast it moves), and shipDead and fired (which tell the alien if the player has died or a bullet has been fired). The alien is controlled by the spaceAlien() function, which starts at about line 100.

Before you look at the detailed code, let's explore what each of the functions that control the aliens are. (You can find them all within the spaceAlien() function.)

alienSpawn()

This function initializes the alien when it first appears on the screen and every time it reappears after being shot.

onEnterFrame

The onEnterFrame event initially calls alienBrain(), which is the main alien-controlling script. When the alien dies, this script is overwritten by alienReincarnate(). This second script is what the dead alien does; it simply waits until it can be respawned.

alienBrain()

This script is the main script that controls the alien. It's a modified version of the swarm script you saw earlier in the book, so although it looks a little complicated, you've already seen most of it. It consists of a few distinct phases.

It first applies the inertia equation to the alien's movement, moving toward the target point (targetX, targetY) in much the same way as the swarm FLA does. If the alien has reached the current target position, the function alienInitialize() is called to decide on where the alien will move next.

It also checks to see whether this alien has been hit by the player's bullet:

```
(this.hitTest(bullet) && fired)
```

and whether this alien has managed to hit the player's ship:

```
this.hitTest(ship)
```

alienReincarnate()

This function runs when the alien has been hit by the bullet. It hides the alien and waits for a random amount of frames. It actually waits until `Math.random()<0.005` or, put another way, it has a 1 in 1/0.005 (1 in 200) chance of respawning from the top of the screen every frame after it has died. On average, each alien will therefore reappear within 200 frames after it has been hit.

To respawn an alien, the `alienReincarnate()` function calls `alienSpawn()` (to reinitialize the alien) and then reattaches `alienBrain()` to the `onEnterFrame` event, causing the reincarnated alien to start living and breathing again. Spooky!

The `alienInitialize()` function handles the alien's decision making. The alien moves in a set of straight-line paths, and this function initializes each path. This function decides whether the alien will move to a random position or whether it will swoop toward the player's ship. The latter is when the alien seems to dive toward the player's ship, and the former is when the alien mills around at the top of the screen.

The aliens become more aggressive as time goes on because the deciding factor between dive and swarm is based on the current skill level:

```
(skill < Math.random())
```

As `skill` goes up, the aliens are more likely to dive toward the ship.

> *Unlike the code for the player's ship, which has to initialize only once per game, the alien code has to initialize the alien many times, which is why you have a separate initialization function for the alien.*

The `alienHit()` function handles the alien being hit. It updates the score and changes the `onEnterFrame` script from `alienBrain()` to `alienReincarnate()`. This has the effect of changing the movement of the alien from "living" to "dead."

`makeAlien()` is a function outside the main `spaceAlien()` template. It is used to . . . well . . . make aliens. It takes an argument, `quantity`, which changes the number of aliens in the game. Setting this argument to a higher value will increase the number of aliens.

> *If you wanted to mimic the days when arcade screens were filled with hordes of aliens or baddies (anyone remember* Gauntlet*?) but your system doesn't have the processing power, you could bitmap cache the baddies. Bitmap caching would work well for our aliens here because their appearance is largely constant (if you don't count the part where they are exploding into a million pieces!). Take a look at the* `spaceAlien()` *function in the code of* `zapper_cacheAsBitmap.fla` *to see where* `cacheAsBitmap` *has been enabled (when the alien has just been created) and disabled (before the alien is due to explode).*

An important (but subtle) point to notice is that makeAlien() will *not* be attached to each instance of the alien sprites. This is because it's outside the spaceAlien() function. Instead, this function is on _root.

OK, back to the detailed code.

The SwarmAlien code

The first surprising thing to notice about the SwarmAlien is that it doesn't actually have to have an instance name for the game to work. This is because each alien doesn't divulge any information to the other sprites, so there's no need to reference it. You give each alien an instance name, though: alien0 to alien9.

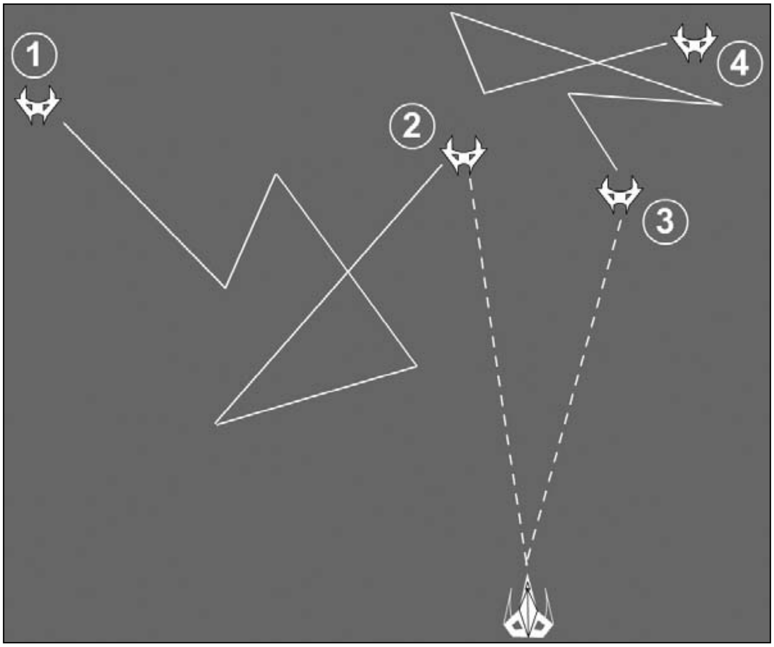
The SwarmAlien has a scary number of local variables. alienX and alienY are its positions on the screen, and startX and startY are its starting positions. The starting positions are used because the SwarmAlien never actually dies. Once it's hit, it waits a short time and respawns at the original start position, which is just offscreen (actually, it's 500 pixels above the visible screen). There are *always* the same number of aliens about at any one time. It's futile to attempt to kill them all, but you'll have fun trying!

distX and distY are variables that you've met before—they're the *where you are* terms in your real motion equation from Chapter 9. targetX and targetY are your *where you want to go* terms. The alienAdvance variable is a modification to the swarm code you saw in Chapter 9, which the aliens will be using. It represents the distance from the top of the screen that the alien is allowed to swarm in. As the game progresses, this variable increases and the alien swarms lower down the screen, making it closer to its target, the ship.

As soon as the alien has been shot, the respawned alien has to start again from the top, and alienAdvance is reset to its start value to make this happen. The secret of the game is for the player to shoot at the lowest aliens, because they will, on average, have the highest alienAdvance, and therefore give the player the biggest headache.

So how does each alien move and attack? Well, you'll be using the swarm code that you saw in Chapter 9. That had a *target* set of variables, which was your *where you want to go* term. Each alien moves much like the flies in the swarming animation you created in the last chapter.

If the alien decides to swarm, your `target` is a random position somewhere in the top `1/alienAdvance` of the screen. `alienAdvance` is 4 to start off with (it's set to this in the function `alienInitialize()`), so you're initially looking at the alien swarming somewhere in the top quarter of the screen, well away from the ship and posing no real threat (the words "sitting" and "duck" come to mind). You'll thus see swarming motion such as the motion shown in the following image between point 1 and point 2, and between point 3 and point 4.



This motion is set up by the first leg of the `if` (highlighted in the following code) in the function `alienInitialize()`. If the `skill` variable (which is between 0 and 1) is less than a random number between 0 and 1, the next path will be defined as one that leads to swarming (i.e., moving to a random position at the top of the screen). As `skill` goes up (which it does as the game progresses), the alien is less likely to swarm and more likely to take the alternative, more aggressive diving motion.

```
this.alienInitialize = function() {
  if (skill < Math.random()) {
    this.targetX = Math.random()* (gSCREEN_RIGHT-gSCREEN_LEFT);
    this.targetY = gSCREEN_TOP+Math.random()* ➥
    (gSCREEN_BOTTOM/this.alienAdvance);
    } else {
      this.targetX = ship._x+((Math.random()*100)-50);
      this.targetY = ship._y-5;
    }
  if (targetY>gSCREEN_BOTTOM) {
    targetY = gSCREEN_BOTTOM+10;
  }
};
```

The else part of the same if is what causes the alien to dive (i.e., movement such as the one shown between points 2 and 3 in the preceding image). If the alien isn't swarming, then you cause it to dive toward the player's ship by making the alien take a path that has a destination that's close to the current player's position.

```
    } else {
      this.targetX = ship._x+((Math.random()*100)-50);
      this.targetY = ship._y-5;
    }
```

The second if caters to the fact that there's no bottom limit to how far down the aliens can advance. If the alien intended to go lower than the bottom of the screen, it's constrained to just above the bottom of the screen. In this way, if the alien intended to move across and behind the ship, it will actually now sweep toward the ship on a collision course:

```
if (targetY>gScreenBottom) {
  targetY = gScreenBottom+10;
}
```

The alienInitialize() function defines only the *where you want to go* part of the alien motion. It doesn't actually cause the animation. This is done by the other major function, alienBrain().

The alienBrain() function is the big one. This is where the modified swarm comes in. You can really see code reuse at work here, because it's essentially the same idea with a few tweaks. The first five lines are almost the same as the ones you saw for swarm, except you add a *1/constant* term that's now 1/accel. As accel gets smaller, the aliens will get quicker and more efficient. The first block of code works out how far away the alien is from where it wants to be and stores this distance as the dist variable. This is then used in the movement equations to create a new position partway toward the target position, alienX, alienY.

```
this.alienBrain = function() {
  this.distX = this.targetX-this.alienX;
  this.distY = this.targetY-this.alienY;
  // Apply inertia equation
  this.alienX = this.alienX+(this.distX/accel);
  this.alienY = this.alienY+(this.distY/accel);
```

The second if looks at whether you're close to the target. If you've reached the destination (rather, if you're within 15 pixels of it), you allow the alien to swarm a bit further down the screen in recognition of its bravery in getting so far by reducing alienAdvance by 0.05. This variable decides which fraction of the screen the alien can swarm in, and reducing its value will increase that fraction.

Sanity check: One-third is less than one-half. As the bottom number decreases, the fraction goes up in value.

You then jump to alienInitialize(), which will set a new target for the alien to move to. This will, on average, be slightly lower down the screen and closer to the ship. In this way, the alien will slowly advance down the screen toward the hapless ship, and it's also more likely to dive the lower it gets.

```
    if ((this.distX < 15) && (this.distY < 15)) {
      this.alienAdvance -= 0.05;
      this.alienInitialize();
    }
```

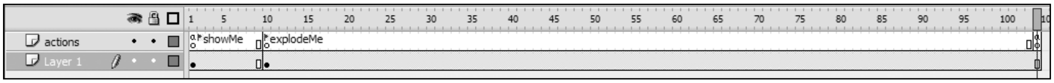
The final part of this block of code performs the sprite movement and checks to see whether the alien has hit either the bullet or ship in its movement. If the bullet and this alien collide, you jump to the alienHit() function and set fired to false (which allows the user to fire again). If you've hit the ship, you want the ship to die (and you communicate this by setting shipDead). For added realism, you could also make the alien explode by going to alienHit(), but don't you think the alien deserves a break for its bravery?

```
    // perform animation
    this._x = this.alienX;
    this._y = this.alienY;
    // check for collisions
    if (this.hitTest(bullet) && fired) {
      fired = false;
      this.gotoAndPlay("explodeMe");
      this.alienHit();
    }
    if (this.hitTest(ship)) {
      shipDead = true;
      this.targetY = 100;
    }
  };
```

The alienHit() function (line 165) is called when the alien is hit. This is a simple script that adds the points for this kill to the score. You get 10 points for each alien, which is multiplied by the current level, so you get more points the longer you last.

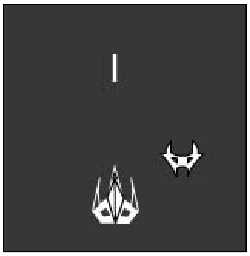
```
  this.alienHit = function() {
    score += level*10;
    score_txt.text = score;
    this.onEnterFrame = this.alienReincarnate;
  };
```

As well as the code, the alien graphic has a timeline (see the following image). Have a look in the Library at symbol `sp.alien`. Normally, the alien's timeline will stay at frame 1. When the alien is hit, the alien timeline will start playing, and you'll see the explosion. `onEnterFrame` is changed to `alienReincarnate()`, which causes nothing to happen for a random period before the alien starts again.

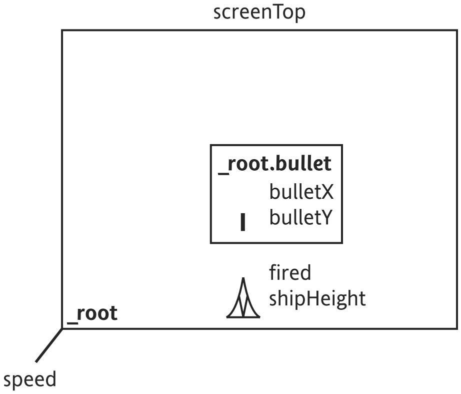


The bullet

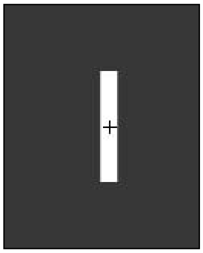
The bullet is what comes out of the player's gun. There's only one bullet in the game—the player is always firing the same one, although it's no less potent with each use. We hope for humanity's sake that your aim is better than the joker's in the following image . . . miles away from the target!



You can tell that the bullet is controlled by some basic code just by looking at its object diagram:



Visually, the bullet is equally bland. It's just a long white rectangle, as you can see in the Library (`sp.bullet`).



When the game starts, you'll see the stationary bullet somewhere just outside the top-left corner of the stage. Although you can see the bullet in Test Movie, you won't be able to see it when you view the SWF in the browser. Press F12 to view the game in your default browser (Cmd+F12 on the Mac).

When the game is running, the bullet just sits on the stage, but it's invisible (its _visible property is set to false). As soon as you fire the laser by pressing the spacebar (thus making fired = true), the bullet is made visible and moved up the screen, starting at the current position of the ship. This will continue until either the bullet makes it to the top of the screen or an alien detects it has been hit by the bullet, in which case the bullet will simply turn itself invisible again and stop. That's all the bullet does!

> *The collision checking between the aliens and the bullet is carried out by the aliens rather than the bullet. See the code under the check for collisions comment in the function* alienBrain() *around line 128. This is easier, because the process then doesn't need to know how many aliens there are. As you add extra aliens, you also add more running scripts per alien to make the collision check, so neither the aliens nor the bullet know how many aliens there are.*

Let's see how this is implemented as code.

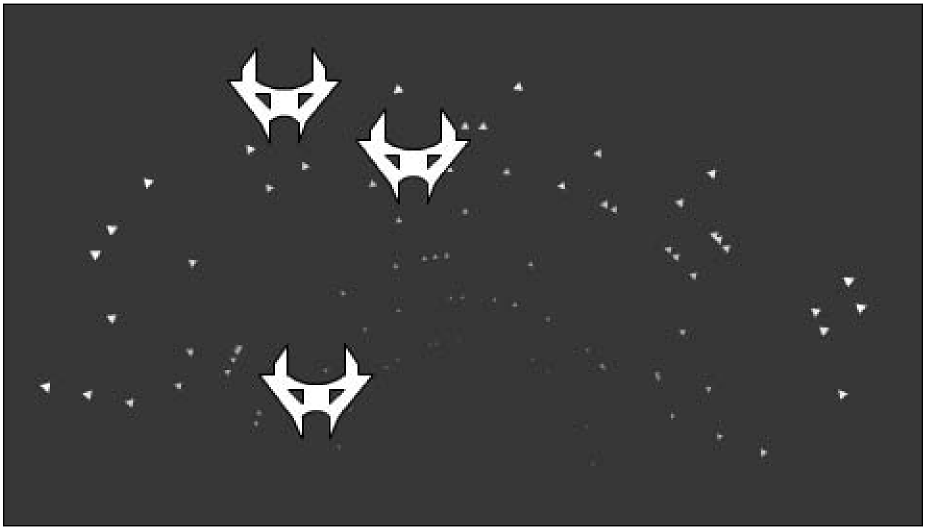
Inside the function bullets(), you'll see an onEnterFrame definition. The code that runs if fired is true (highlighted in the following listing) is the main bullet animation code. It will move the bullet up at three times speed until the bullet gets to the top of the screen (this.bulletY < gSCREEN_TOP) or fired becomes false (which will occur if an alien detects it has been hit via the alienBrain() function). The else part of the if (fired) is run if the bullet is *not* currently fired (fired == false), and it turns the bullet invisible.

After the end of the bullet's onEnterFrame code, you'll see a couple of lines setting bulletX and bulletY to 0, which is the position the bullet will take at the start of the game (at the top-left corner of the stage).

```
function bullets():Void {
  //
  // Function to animate bullet...
  this.onEnterFrame = function() {
    if (fired) {
      this._visible = true;
      this.bulletY = this.bulletY-speed*3;
      this._y = this.bulletY;
      if ((this.bulletY < gSCREEN_TOP) || (fired == false)) {
        fired = false;
        this.bulletY = gSCREEN_BOTTOM-gSHIP_HEIGHT;
      }
    } else {
      this._visible = false;
    }
  };
  this.bulletX = 0;
  this.bulletY = 0;
};
```

The debris of war

No laser battle in space is complete without some level of pyrotechnics. This game includes separate explosions for the aliens and the ship. The following image shows the SwarmAliens victoriously moving through the expanding plasma cloud that was once a spacecraft.



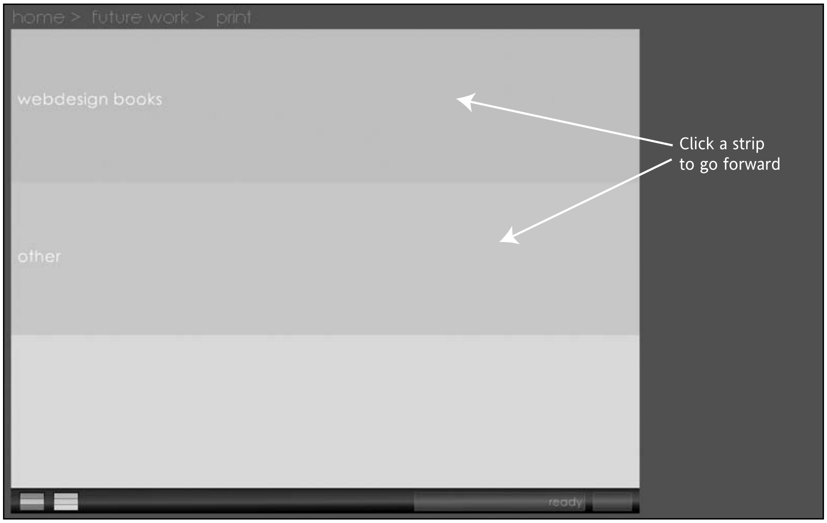
This is all about the scrolling star field that's the backdrop for the battle. No space game is complete without one. The star field is controlled by the `starDot()` and `makeStar()` functions. These are just very dumbed-down versions of the alien scripts.

Now that you've had a look through the code for the zapper game, take a few minutes to play the game. Go on, pretend you have a stack of quarters in your hand and you're back in the video game arcade you used to spend all your time in when you were a kid. Once you've played the game, think about how relatively simple the ActionScript that made it possible is.

Book project: Navigation event handling

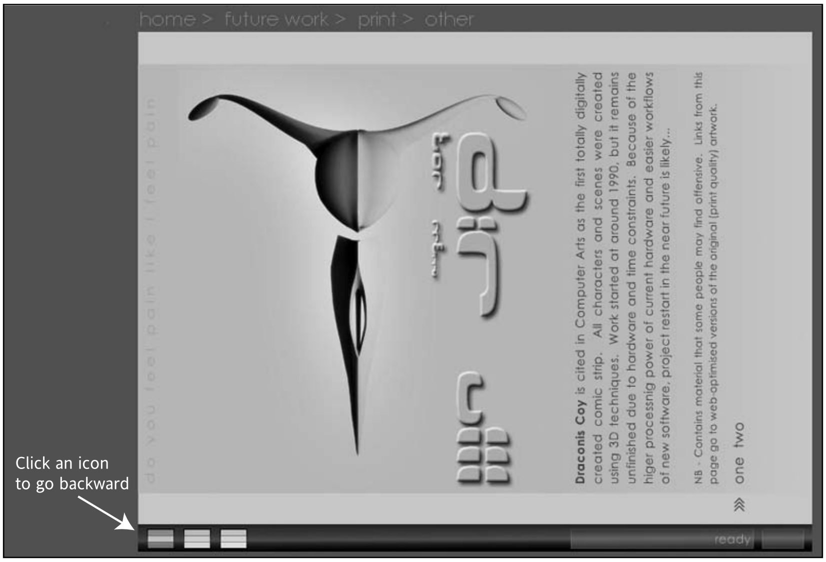
You've nearly reached the end of the case study and you've learned an awful lot from it. We hope you realize just how far you've come. There are just three parts of the Futuremedia site left to take care of.

- **The forward navigation**, which is what happens when you click one of the colored strips



Click a strip to go forward

- **The reverse navigation**, which is what happens when you click one of the icons (at the bottom of the screen) to move backward


home > future work > print > other
do you feel pain like i feel pain
one two
ready

Draconis Coy is cited in Computer Arts as the first totally digitally created comic strip. All characters and scenes were created using 3D techniques. Work started at around 1990, but it remains unfinished due to hardware and time constraints. Because of the higer processnig power of current hardware and easier workflows of new software, project restart in the near future is likely...

NB - Contains material that some people may find offensive. Links from this page go to web-optimised versions of the original (print quality) artwork.

Click an icon to go backward

- **The content** you'll need to add to the site

In this section, you'll begin looking at the navigation.

There's no hiding the fact that the code in Futuremedia will start getting a little tricky from here onward.

Unlike site design projects you may have seen in other books, in this book project my intention has been to avoid the "now put this bit of code at this keyframe without really knowing why you need to do this" teaching method or, even worse, the "this site is so easy that it needs no explanation (which also makes it totally useless because it teaches nothing about ActionScript)" school of thought. Instead, we've tried to explain the thought process along with the design to show the "why" behind the design, and we've also used a moderately complex Flash ActionScript site.

So, as the code gets more and more esoteric, you'll see more and more sketches and asides. They aren't there to simplify the code for you. They're there because they're the tools we used to actually write the code. An understanding of the sketches is more important than an understanding of the code, because the former will teach you to think like an ActionScript coder, whereas the latter will only enable you to type in the code for the Futuremedia site.

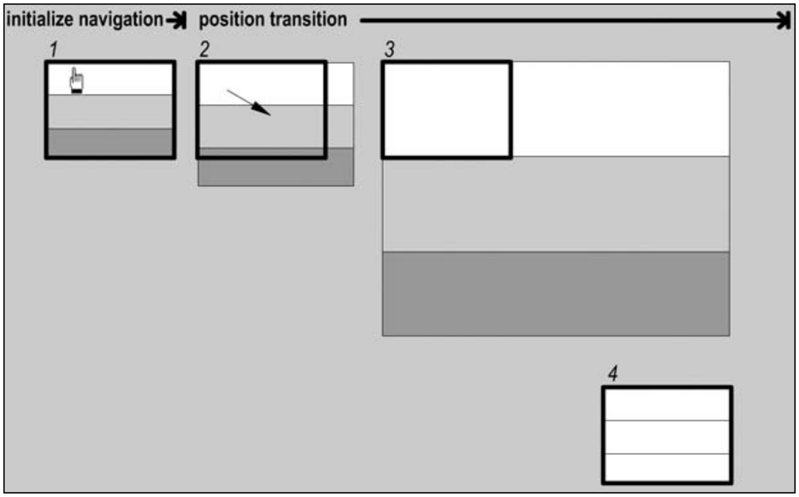
Sanity check #1

Because you'll be playing around with a lot of code, there's a full printed listing of the final FLA at the end of this project section. You'll also find two FLAs that contain the working files, `futuremedia_code01a.fla` and `futuremedia_code02.fla`, at the midpoint and end of this part of the project in the download for this chapter.

Adding the basic UI animation

In the last section of the project you ended your work on the `DEFINE EVENT SCRIPTS` part of the listing with a simple script that returned which strip you clicked. You'll have a look at extending this and creating the first, very basic event-driven code to start animating the UI.

In the process, you'll take a look at how we started planning this via **UI storyboards**, little sketches that gave us an idea of how the interface should move and how to achieve this movement. The storyboards are the actual FLAs used early on in the design to visualize how to convert the ideas into code. Here's a complete example:



This proves that coding is a visual process. The code was written to a set of drawings, and this is true of commercially written code in anything from long-range missile global positioning/navigation systems with low-level flying Nap of the Earth (NOE) digital autopilots to all but the simplest web scripts.

If you didn't look at the diagrams and you just went straight to the code for Futuremedia, you'd probably just look at it and think, "Whoa! Where do I start with this?" With the diagrams, you should start to see the intent behind the site and, more important, you should see how the code is just a symbolic representation of the drawings that the computer can understand. Coding and drawing are closer than most graphic designers may suppose—it's just a well-kept secret!

So far you have the three strips outputting a message when you click them. What you actually want to do is something like this:

1. When you click a strip, you need to **initialize the navigation**. This is what the current navigate function will eventually do (step 1 in the storyboard). You then need to animate the navigation. You can split this into two separate parts.

2. The first transition is the zoom/pan **position transition**. This is where the selected strip zooms and pans so that it fills the screen (step 3 in the storyboard). There's actually a trick at the end where the three strips flick back to their original positions as well, and this involves a color change so that all three strips take on the selected strip's color and look the same as a single strip on its own (step 4 in the storyboard). This step requires you to be really sneaky, because the user isn't supposed to see it!
3. You need to fade the strips into three new colors via a **color transition**, and this takes you back to step 1. If you want to display content at step 4 rather than present further navigation options, you simply leave the three strips as the same color, as shown in step 4 in the storyboard.

Let's build the core events that will handle these transitions. At the moment, you should have a file that contains the following code under the comment DEFINE EVENT SCRIPTS at the top of the script (you can also look at the file futuremedia_code01.fla from the last chapter).

```
// DEFINE EVENT SCRIPTS
function navigate():Void {
  // initialize navigation event handler here
  trace("");
  trace("you pressed me and I am");
  trace(this._name);
  trace("but you need to fully define me");
  trace("before I start doing anything else!");
}
```

At the moment, navigate is a button event script. It will run only once, which isn't useful for the smooth animations you want. What you'll do instead is use navigate() to start *other* events. What do you need these other events to do? Well, you need

- An onEnterFrame that performs the zoom and pan effect
- An onEnterFrame that performs the color fade effect once the zoom and pan effect has finished

Yeah, you probably thought the site zooms and fades at the same time, but if you look at the finished site carefully, you'll see that the color fading doesn't happen until *after* the zoom.

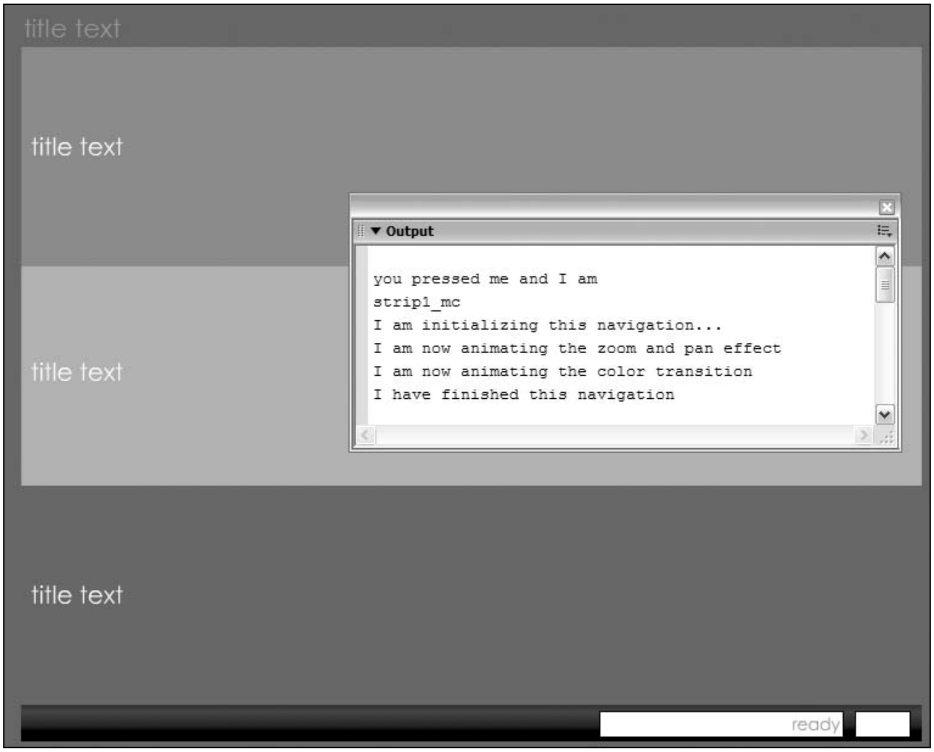
> *A button event that starts up a whole series of other events running isn't something new. You did exactly the same thing when you created the* click here to go *button in the zapper game.*

At the moment, navigate() is a simple stub that doesn't do anything. You'll stick to the same stub structure, but you'll expand navigate() to include your zoom/pan and color transition animations. Delete the current navigate() function and replace it with the following code:

```
// DEFINE EVENT SCRIPTS
function navigate():Void {
  // initialize navigation here
  trace("");
  trace("you pressed me and I am");
  trace(this._name);
  trace("I am initializing this navigation...");
  // then start navigation graphical transition...
  this.onEnterFrame = posTransition;
}
function posTransition():Void {
  // now do the actual animation...
  trace("I am now animating the zoom and pan effect");
  // we have now done the position transition...
  delete this.onEnterFrame;
  // we next do the color transition...
  tricolor.onEnterFrame = colTransition;
}
function colTransition():Void {
  // We finally do the color transition that makes the tricolor split
  // back into 3 here
  trace(“I am now animating the color transition”);
  // finally, remove all event scripts...
  trace("I have finished this navigation");
  delete tricolor.onEnterFrame;
  trace(""), trace(""), trace("");
}
```

If you test this FLA, you still won't see any animation, but all the steps in the storyboard are being output to the screen in the order in which you want them. Each event handler runs, calls the next one in the sequence, and traces what it should eventually be doing to the Output window.

```
you pressed me and I am
strip1_mc
I am initializing this navigation...
I am now animating the zoom and pan effect
I am now animating the color transition
I have finished this navigation
```



```
you pressed me and I am
strip1_mc
I am initializing this navigation...
I am now animating the zoom and pan effect
I am now animating the color transition
I have finished this navigation
```

You've arranged the events in the correct order before you start adding the actual functionality. This is an important feature of using functions—you can add them as placeholders to define the structure. These placeholders define a basic framework (or skeleton, or you might even want to think of them as empty black boxes) for your final code. Once you have the framework right, you can go ahead and look at each function in isolation, fleshing (or should that be "Flashing"?) out the code as you add meat to the skeleton.

So you now have a skeleton. The next step is to define each event so that it starts moving things. In each function, the process is the same:

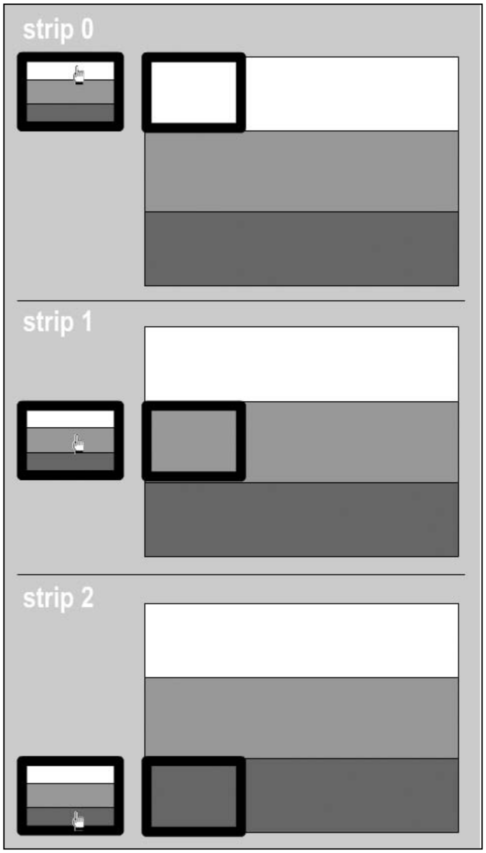
1. Draw or otherwise visualize what it is you want to happen.
2. Convert it into code.

navigate() and posTransition()

You'll add the meat to the two stub functions next. You're better off doing them in the order in which they'll run.

First up, the button event, `navigate()`. `navigate()` has to initialize the animation. It has to define the start and end position of the animation. Let's call the start position (`x, y`) and the end position (`targetX, targetY`). To initialize, you have to know what (`x, y`) and (`targetX, targetY`) are in terms of things you understand, and then in terms of things your computer understands.

First, you need to visualize this graphically.

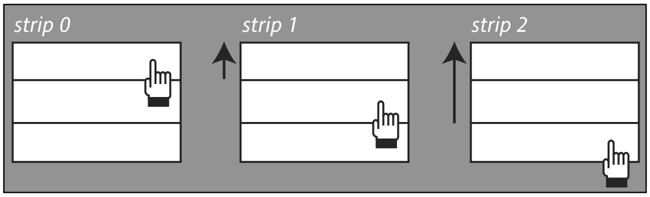


In each strip, the small image is what the site looks like before you click a strip, and the large image is what the site will look like after the position transition has occurred. The solid black frame is the visible screen area. It looks like an IQ test, and that's what it probably is. See how well you do . . .

Question 1: What is the relationship between the three small images and the three big images next to them, expressed in terms of anything you can see in the picture?

No peeking! You have one minute. OK, time's up.

Answer 1: When you click any strip, the top-left corner of that strip moves to the top-left corner of the visible screen area. As it does this, the `tricolor` *clip trebles in size.*



Question 2: For an extra 3 points (and the game), express the transition mathematically in terms of what you already know.

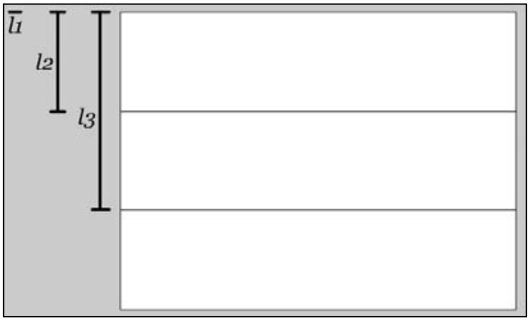
Now it gets a bit trickier! At the risk of stating the obvious, you know the transition is a position-based one, but less obviously, you also therefore have to express the answer in terms of positions you already know only.

The starting position of `tricolor` is the top-left corner (`border, border`). At the risk of stating the obvious again (we have to, because we're simply reiterating the thought process when designing this site to provide insight), the final position must be *related* to something that has to do with the three strips, because that's the only known thing that changes. You can glean half the answer fairly quickly by looking at the existing drawings so far. The tricolor has to move a distance equal to the distance `l` (lowercase "L").

- If you click `strip0_mc`, the distance `l` (shown as `l1` in the following diagram) is 0.
- If you click `strip1_mc`, the distance is `l2`, and if you click `strip2_mc`, the distance is `l3`.
- This distance is always actually the same—it's the *distance the selected strip is from the top-left corner*.

It took a while to work this out, because one of the lengths was actually 0; the top strip is already at the top-left corner point.

> *The moral of this story is that 0 is never a special case. It's a number just like any other, and if your logic assumes it's a special case, it's probably because your logic is flawed.*



The distance of the top-left point of any given strip to the corner point you want to move it to is *always*

```
-this._y
```

where this is the strip you've selected *inside* tricolor, and you're taking into account the *direction* you have to move as well (that's what the - sign does). However, the tricolor is also scaled up by 300% so that a single strip grows enough to fill the screen. You therefore also have to increase this length by the same amount.

```
-3*this._y
```

You'll use this to move tricolor rather than the selected strip (so that all three strips move rather than just one), and you also need to take into account the fact that you don't want to move to the top-left corner of the screen, but to the border:

```
(-3*this._y)+border
```

or simply

```
-3*this._y+border
```

Note that you don't need to worry about the _x position—the elements are already at the required x position.

So you have your answer.

Answer 2: You already know which strip you've selected from the last chapter, because the script prints this in the Output *window every time you click a strip. If you call this strip* this, *then the position you want to move to, relative to the center point, is as follows:*

```
-3*this._y+border
```

And you also need to scale the movie clip containing the three strips, tricolor, *from 100% to 300%.*

Now replace the navigate() function with this:

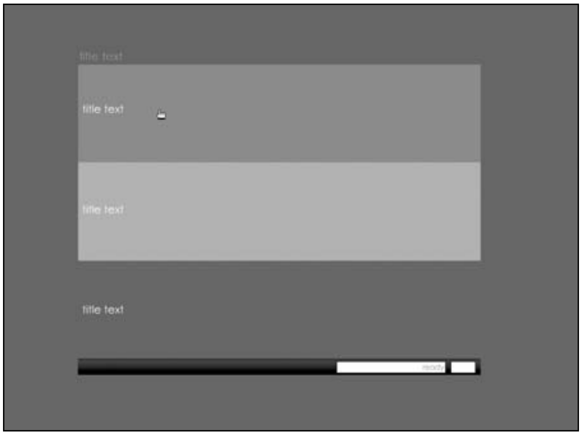
```
function navigate():Void {
  // set up target for animation transition...
  scale = 100;
  x = BORDER;
  y = BORDER;
  yTarget = -3*this._y+BORDER;
  scaleTarget = 300;
  // attach the animation transition to the strip
  // that has just been clicked...
  this.onEnterFrame = posTransition;
}
```

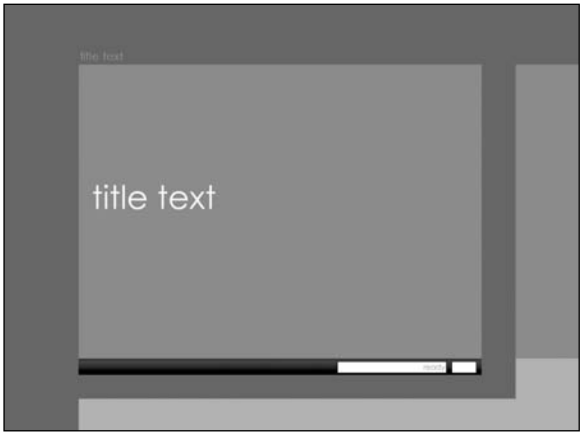
Replace the posTransition() function with this:

```
function posTransition():Void {
  scale = scaleTarget;
  y = yTarget;
  tricolor._y = y;
  tricolor._xscale = tricolor._yscale = scale;
}
```

When you click a strip, that strip fills the screen. You can do this only once at the moment (as soon as you click a strip, the other strips disappear offscreen, so run the SWF three times, clicking one of the three strips each time). Don't worry about all the stuff that appears off the stage area. If you test the site in the browser (F12/Cmd+F12), you'll see that this content isn't displayed.

The following images show the before and after when a user clicks the top strip, strip0_mc.





To keep things simple, you're developing only the forward navigation to start with; you won't be able to go backward yet. At this stage, you'll have to run the FLA again if you want to test navigating forward from a different starting point. To create the ability to go backward through the navigation requires you to be able to store the path you take when moving forward, and this means that you need to make sure you have the forward bit sorted out first.

So now that you've run this (as yet still basic) animation, let's have a look at the code you just added. navigate() sets up your starting position (y) and size (s), and your target destination (yTarget) and size (sTarget). navigate() is your onRelease event, and this starts up the posTransition() animation. posTransition() simply makes tricolor go to the final destination in one shot.

Adding typing

You've just added a load of variables without really defining them properly, so let's do that now. In the main code, add the following highlighted code (this code section is toward the end of the script).

```
// Define navigation strip variables...
var NAVSTRIP_X:Number = BORDER;
var NAVSTRIP_Y:Number = BOTTOM;
// Define animation variables
var scale:Number = 0, x:Number = 0, y:Number = 0;
var yTarget:Number = 0, scaleTarget:Number = 0;
// Initialize UI...
stripPage.apply(tricolor);
```

Sanity check #2

Since you've added a lot of code so far, we've created a working intermediate FLA for those who aren't seeing the effect so far (or who just want to check that they're seeing the right thing). Have a look at futuremedia_code01a.fla if you have any worries.

For those having trouble with the code so far, here are a few words that might make more sense of the situation. scale is the current scale factor. At the moment, you set it to 100% in navigate() and 300% (which is the same as scaleTarget) in posTransition(). That, on the face of it, looks a bit silly. Why not forget scale and just use scaleTarget? Well, scale will be varied between 100% and 300% *gradually* in the final site design, and this is what gives you the smooth transition. You'll look at creating a smooth transition next.

Creating a smooth transition

At the moment you have a single-step animation. As soon as you click a strip, the clicked strip scales to its final size immediately.

Let's slow it down a little. In this section you'll add the inertia effect to the scale. This is very easy to do, because you've already set up all the variables you need. Rather than make scale and y equal to the *where you want to go* values scaleTarget and yTarget, you make them change via an inertial effect in the movement function, posTransition(). Change the following lines currently in posTransition():

```
scale = scaleTarget;
y = yTarget;
```

to the following:

```
function posTransition():Void {
  scale -= (scale-scaleTarget)/3;
  y -= (y-yTarget)/3;
  tricolor._y = y;
  tricolor._xscale = tricolor._yscale = scale;
}
```

OK, you now need to do a little trick to put the tricolor back to the starting position without anyone knowing. In the function posTransition(), add the following lines:

```
function posTransition():Void {
  scale -= (scale-scaleTarget)/3;
  y -= (y-yTarget)/3;
  tricolor._y = y;
  tricolor._xscale = tricolor._yscale = scale;
  if (Math.abs(scale-scaleTarget)<5) {
    // if so, disconnect this function from onEnterframe...
    delete this.onEnterFrame;
    // then place tricolor back at start position...
    tricolor._y = BORDER;
    tricolor._xscale = tricolor._yscale = 100;
  }
}
```

This returns tricolor to its original position, but it's a little obvious to say the least! What you need to do is find out the color of the selected strip and apply it to all three strips. The color of each strip, stripn_mc, where n equals 0, 1, or 2, is held in variable colorn, where n equals 0, 1, or 2.

Obviously, then, n is the key. To get n from the name of the selected strip, you use the following method of the String object:

```
substring(5, 6)
```

You apply that to the name of the current strip:

```
selectedStrip = this._name.substring(5, 6);
```

This will return 0, 1, or 2.

To form the required colorn variable (or, more specifically, _root.colorn), you use this:

```
selectedCol = _root["color"+selectedStrip];
```

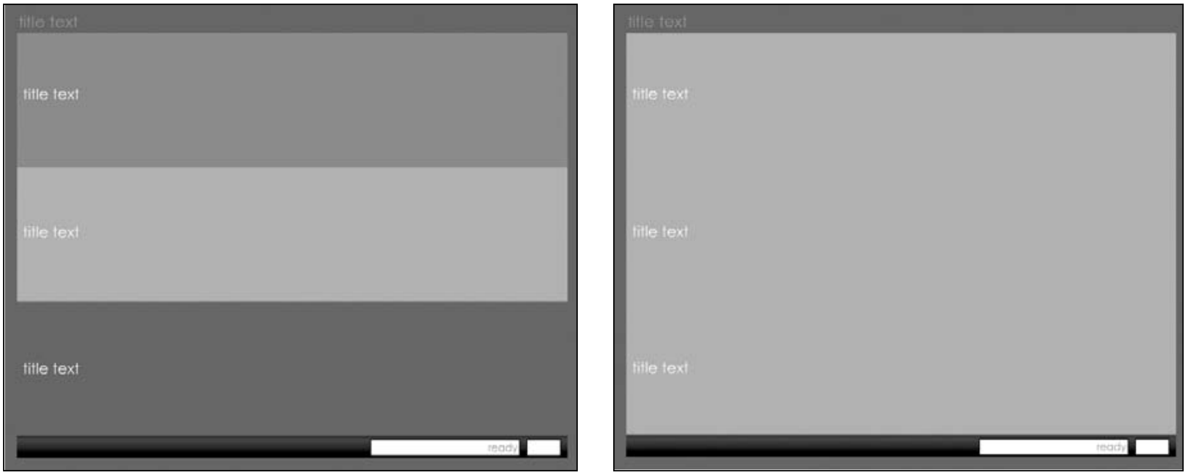
If selectedStrip is 2, _root["color"+selectedStrip] will give you _root.color2.

OK, you're almost there. Add the following code to posTransition():

```
function posTransition():Void {
  var selectedStrip:String;
  scale -= (scale-scaleTarget)/3;
  y -= (y-yTarget)/3;
  tricolor._y = y;
  tricolor._xscale = tricolor._yscale = scale;
  if (Math.abs(scale-scaleTarget)<5) {
    // if so, disconnect this function from onEnterframe...
    delete this.onEnterFrame;
    // then place tricolor back at start position...
    tricolor._y = BORDER;
    tricolor._xscale = tricolor._yscale = 100;
    selectedStrip = this._name.substring(5, 6);
    selectedCol = _root["color"+selectedStrip];
    tricolor.setCol(selectedCol, selectedCol, selectedCol);
  }
}
```

If you test the FLA now, you'll see the color of all three strips change on the snapback. You're calling the previously defined function setCol(). You used this in the last chapter to turn the gray strips to the three colorful ones.

In the following image, you can see before and after images of a user clicking the middle strip.



The title text text on each strip gives the game away a little, but you can soon fix that by hiding it.

You'll fade the text while the transition occurs. To do this, you need a value that goes from 100% alpha to 0% as the transition takes place. You already have one that goes from 100% to 300%, and that's the scale value that controls scale. It's a good idea to follow the scale value, turning it to what you want, because then your text fade will be in synch with the transition itself.

To change the 100% to 300% of scale to 100% to 0% of your new variable (say fadeAlpha), you use this:

```
fadeAlpha = 300-scale;
```

You also need to change all three text fields' _alpha values according to the preceding equation, so let's create a function to do this. If you're hiding the text, you'll more than likely have to unhide it again later, so you might as well make your code reusable!

Add the following new lines in the function posTransition():

```
function posTransition():Void {
  var selectedStrip:String;
  scale -= (scale-scaleTarget)/3;
  y -= (y-yTarget)/3;
  fadeAlpha = 300-scale;
  tricolor._y = y;
  tricolor._xscale = tricolor._yscale = scale;
  fadeText(fadeAlpha);
  if (Math.abs(scale-scaleTarget)<5) {
    // if so, disconnect this function from onEnterframe...
    delete this.onEnterFrame;
    // then place tricolor back at start position...
    tricolor._y = BORDER;
    tricolor._xscale = tricolor._yscale = 100;
    selectedStrip = this._name.substring(5, 6);
    selectedCol = _root["color"+selectedStrip];
    tricolor.setCol(selectedCol, selectedCol, selectedCol);
  }
}
```

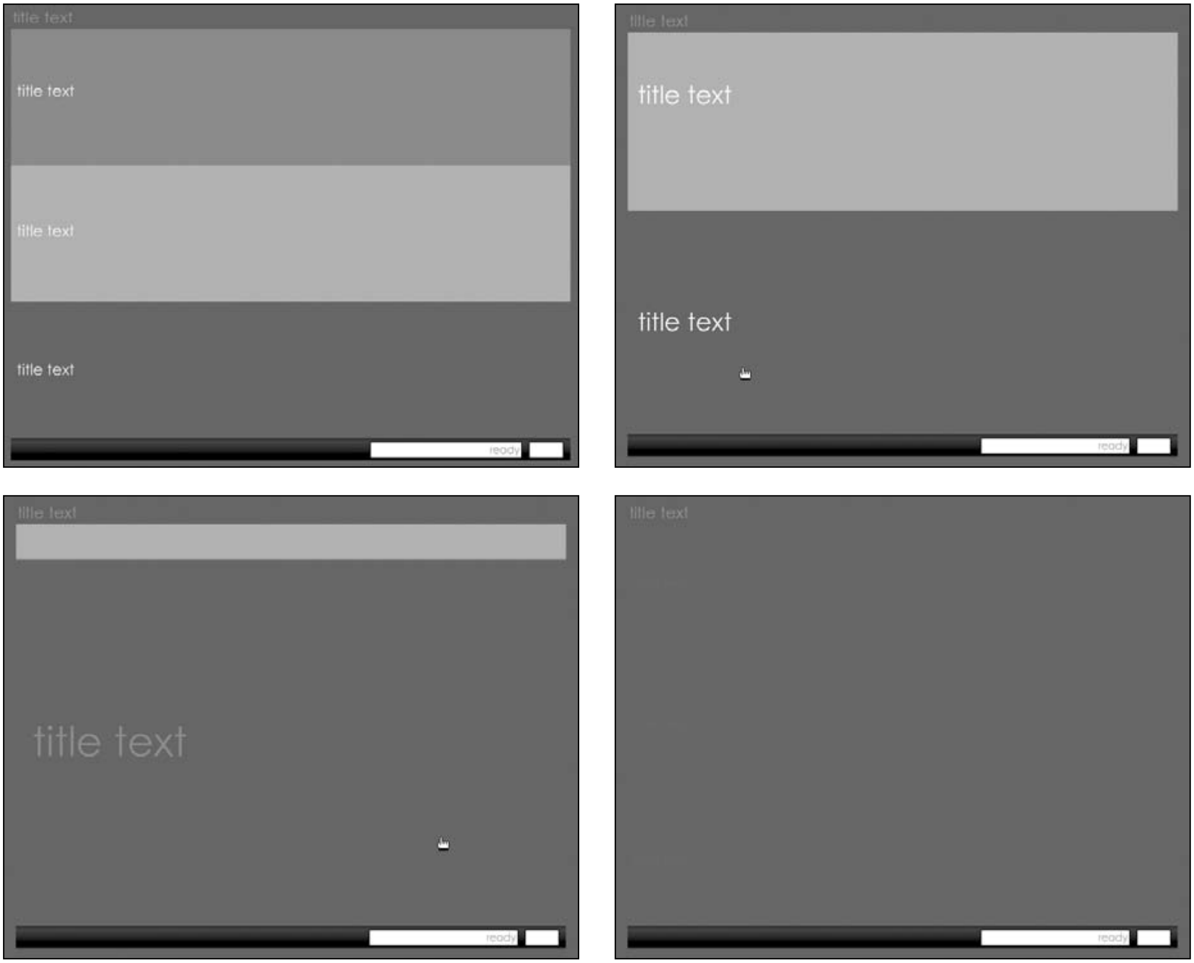
Here you use a new function called fadeText() with the new variable fadeAlpha (this will be declared in a moment with the other animation variables). You still need to define the function, though. After the end of colTransition(), add the new function fadeText() as shown:

```
function fadeText(fadeValue):Void {
  tricolor.strip0_txt._alpha = fadeValue;
  tricolor.strip1_txt._alpha = fadeValue;
  tricolor.strip2_txt._alpha = fadeValue;
}
```

If you run the FLA now, you'll see that your sneaky switch is successful. You should test the site in a browser to see the full deception (press F12/Cmd+F12), and then run it in normal test mode (Ctrl+Enter/Cmd+Enter) to see what's going on behind the scenes. If you still don't believe us, change the frame rate to 1 fps and then watch the transition.

Here's what you'll see. You're first presented with the start screen, the by now familiar orange, green, and blue page.

As soon as you click any strip, you'll see the interface slowly morph in a smooth, organic transition until the whole center area becomes the color of the strip you just clicked. The FLA for this point is included as futuremedia_code02.fla and implements the storyboards so far.



Before we move on, below is a full listing of the site so far. Add the fadeAlpha variable where highlighted.

```
// DEFINE EVENT SCRIPTS
function navigate():Void {
  // set up target for animation transition...
  scale=100;
  x=BORDER;
  y=BORDER;
  yTarget = -3*this._y+BORDER;
  scaleTarget = 300;
  // attach the animation transition to the strip
  // that has just been clicked...
  this.onEnterFrame = posTransition;
}
```

```
function posTransition():Void {
  var selectedStrip:String;
  scale -= (scale-scaleTarget)/3;
  y -= (y-yTarget)/3;
  fadeAlpha = 300-scale;
  tricolor._y = y;
  tricolor._xscale = tricolor._yscale = scale;
  fadeText(fadeAlpha);
  if (Math.abs(scale-scaleTarget)<5) {
    // if so, disconnect this function from onEnterframe...
    delete this.onEnterFrame;
    // then place tricolor back at start position...
    tricolor._y = BORDER;
    tricolor._xscale = tricolor._yscale = 100;
    selectedStrip = this._name.substring(5, 6);
    selectedCol = _root["color"+selectedStrip];
    tricolor.setCol(selectedCol, selectedCol, selectedCol);
  }
}
function fadeText(fadeValue):Void {
  tricolor.strip0_txt._alpha = fadeValue;
  tricolor.strip1_txt._alpha = fadeValue;
  tricolor.strip2_txt._alpha = fadeValue;
};
function colTransition():Void {
  // We finally do the color transition that makes the
  // tricolor split back into 3 here
  trace("I am now animating the color transition");
  // finally, remove all event scripts...
  trace("I have finished this navigation");
  delete tricolor.onEnterFrame;
  trace(""), trace(""), trace("");
}
// DEFINE MAIN SETUP SCRIPTS
function stripEvents():Void {
  // attach button events to the 3 pages...
  this.strip0_mc.onRelease = navigate;
  this.strip1_mc.onRelease = navigate;
  this.strip2_mc.onRelease = navigate;
}
function stripCols():Void {
  this.colStrip0 = new Color(this.strip0_mc);
  this.colStrip1 = new Color(this.strip1_mc);
  this.colStrip2 = new Color(this.strip2_mc);
  this.setCol = function(col0, col1, col2) {
    var col0, col1, col2;
    this.colStrip0.setRGB(col0);
    this.colStrip1.setRGB(col1);
    this.colStrip2.setRGB(col2);
  };
}
```

```
function stripPage():Void {
  // note use of Function inheritance
  this.inheritEvents = stripEvents;
  this.inheritCols = stripCols;
  this.inheritEvents();
  this.inheritCols();
}
// Define screen extents for later use...
Stage.scaleMode = "exactFit";
var BORDER:Number = 30;
var BOTTOM:Number = Stage.height-BORDER;
var TOP:Number = BORDER;
var RIGHT:Number = Stage.width-BORDER;
var LEFT:Number = BORDER;
Stage.scaleMode = "noScale";
// Define initial screen colors
// (further colors will use these colors as their seed)
var color0:Number = 0xDD8000;
var color1:Number = 0xAACC00;
var color2:Number = 0x0080CC;
// Define navigation strip variables...
var NAVSTRIP_X:Number = BORDER;
var NAVSTRIP_Y:Number = BOTTOM;
// Define animation variables
var scale:Number = 0, x:Number = 0, y:Number = 0;
var yTarget:Number = 0, scaleTarget:Number = 0;
var fadeAlpha:Number;
// Initialize UI...
stripPage.apply(tricolor);
// Set UI colors...
tricolor.setCol(color0, color1, color2);
// now sit back and let the event scripts handle everything ;)
stop();
```

Parting shots

You still have a pretty unfinished site, but you're moving forward at a good pace now. You only need to bring in the colTransition() event script to complete the main site graphic transitions.

The use of diagrams that define how the code works and what it actually needs to do is one of the cool things about programming in Flash. You don't put away the drawing pencils when you start coding. Most of the time you'll need to sharpen them and get ready for half an hour of playing around and experimenting on paper before you try out your ideas in ActionScript and the Flash environment.

This can be an exciting and rewarding pastime in itself because, unlike with traditional graphic design, with Flash you don't end up with pretty logos and typography—you end up with things that *move*.

At some point, the navigation will have to display pages of content as well as simply linking to new pages, but you'll come to that part in due course. At the moment you're really only concerned with getting the main concept working.

That's where Flash is totally different from static media such as bitmaps or HTML, and if you're just looking over the diagrams and sketches in this section, you'll never get the feel of Flash. It's all about motion and saying, "Look at that move! Cool—this is going somewhere!"

Summary

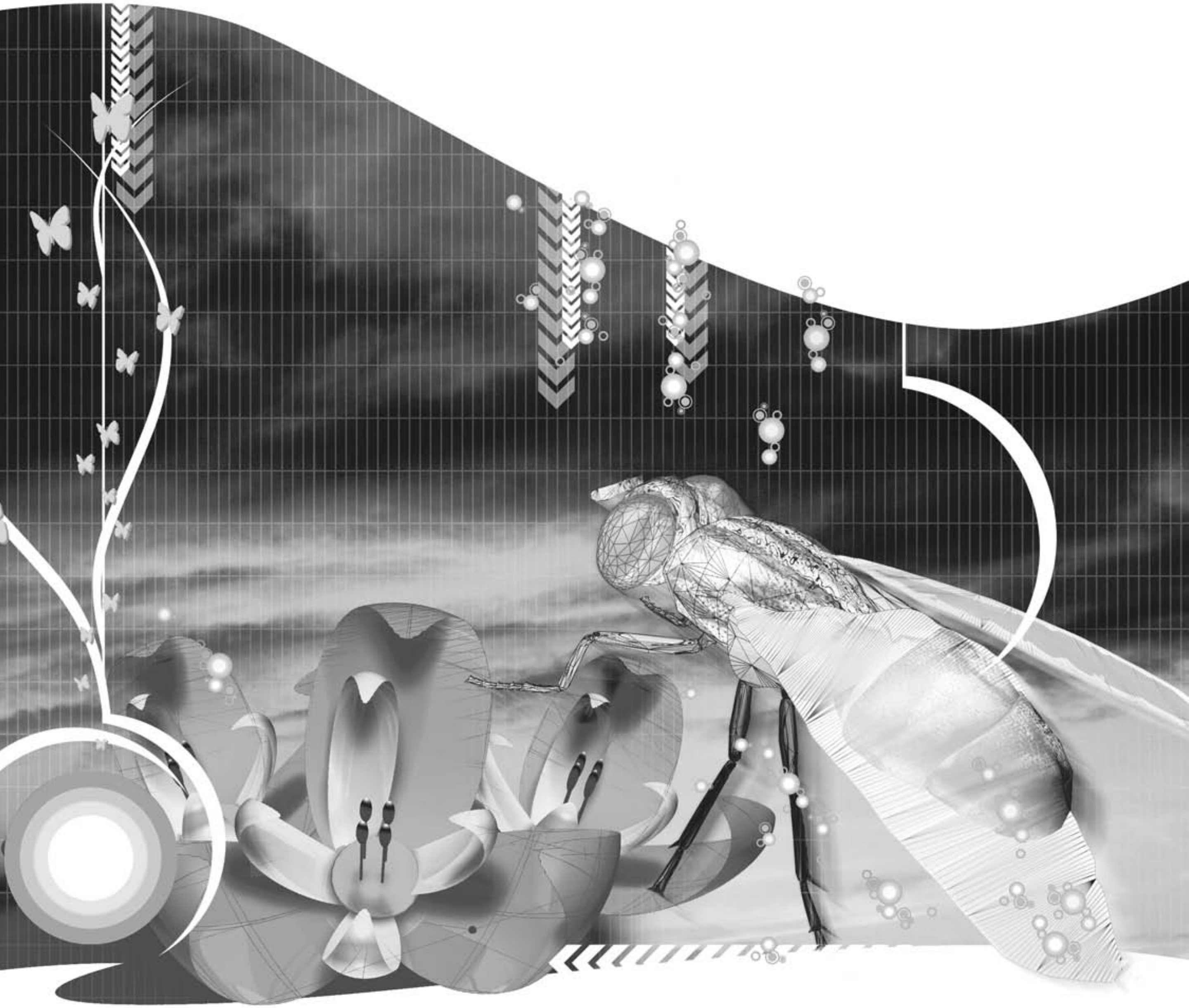
Once you have your head around the way ActionScript works, you'll see that even complicated-looking projects (like the zapper game) are quite simple when you break them down into their constituent parts.

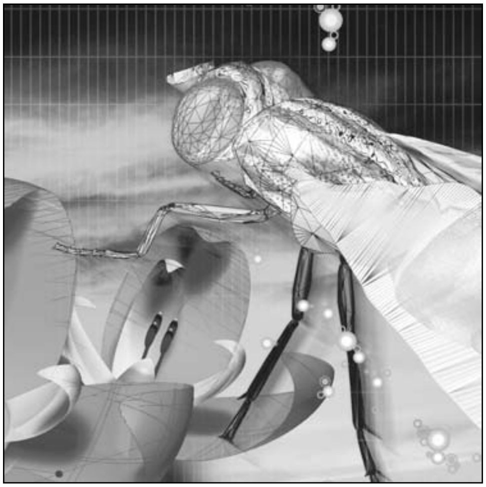
You've seen the same thing in the Futuremedia project, where you've concentrated on one function (`posTransition()`), leaving all the other code alone while you got this one code module working. Although the Futuremedia script is still under 100 lines long, think how much more difficult it would be to add things to the code now that it has grown in length beyond a little trick FLA! All that careful timeline and graphic planning and construction is really paying off now, because you don't have to worry about seeing showstopper problems appearing out of the blue as you build the site, and your use of functions to modularize the code keeps everything manageable.

As well as covering the practical use of code, this chapter also delivered a philosophical message. Coding is not just about knowing what to do. You saw how we developed ideas as sketches before starting to code, and the harder the code became, the more sketches we produced.

Code is never created in isolation. For every ten lines of code, there's probably a rough sketch or diagram somewhere. If you've ever looked at scripts created by some of the masters and thought "Wow, I could never do that!" you're probably beginning to realize that the written code is only a small part of the process. *That's why code appears hard, not because it actually is hard*! The way you've developed the ideas behind the code by finding clever ways around logical and practical problems in a graphic way is the way forward. Code writing is a *creative process*, and as creative designers, you should really be as at home with ActionScript `if` statements as you are with the Flash Pen tool or Photoshop Quick Masks.

The main point we'd like you to take away from this chapter is that you've moved from easy, beginner-level tasks to more advanced tasks you would probably never have contemplated. *You can now do these tasks on your own*. Think how far you've come since you learned about the `stop()` command at the beginning of the book! This is the beauty of ActionScript. It's not really that difficult, and yet you can achieve the most amazing things with it when you think in terms of code *and* the concepts behind what you're trying to achieve.





Chapter 11

DRAWING API

What we'll cover in this chapter:

- Using the drawing API in Flash to dynamically create lines, curves, fills, and pattern effects
- Working with scripted color
- Understanding data-driven sites
- Creating the color transitions for the Futuremedia case study
- Defining the data structure for the case study

One cool feature of Flash is the **drawing API**, which allows you to create new content within your movie clips. You can now go straight to the engine room of the movie clip and produce new sketches *on the fly*. In fact, you can create three things:

- Lines
- Curves
- Fills

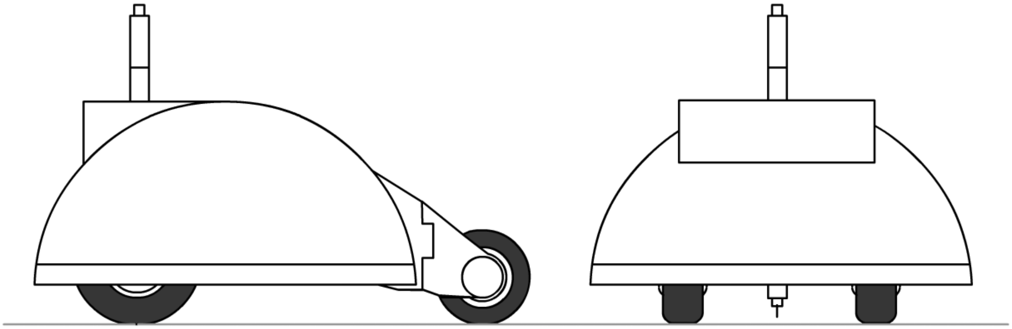
Although you could emulate lines in early versions of Flash, you couldn't control the movie clip with clever scripting to create controlled curves or fills. And it all started with a turtle.

Turtle graphics

Flash's drawing API is based on something called **turtle graphics**. In this section, you'll take a quick look at what that is before getting down to business, because the drawing API can get a little complicated.

Turtle graphics is an old graphics language used way back when to control a little robot with a pen attached to it. The turtle was the robot, which typically looked like an upturned bowl on wheels (hence the name). The robot could do a few basic things:

- Lift the pen up (*penUp*).
- Put the pen down (*penDown*).
- Rotate without moving forward.
- Travel forward (but not at the same time as rotating, so it could travel only in a straight line).



The turtle looked much like the preceding drawing. The pen fit between the middle of two big wheels at the front, and the wheel at the back was like that of a shopping cart: it could twist on a hinge, allowing the turtle to turn on the spot. Because the pen was situated at the center of rotation between the two front wheels, the turtle could rotate without moving the pen tip, and if the turtle moved forward with the pen down, it left a line describing its path.

The turtle had two basic commands:

- *move(x, y)*: This was a *penUp* followed by a rotate until (x, y) was directly in front of the turtle. The turtle would then move forward in a straight line as far as (x, y). This command didn't result in a line being drawn.
- *draw(x, y)*: This consisted of rotating until (x, y) was in front of you, followed by a *penDown* and moving forward in a straight line as far as (x, y). This command resulted in a line being drawn from the start point to (x, y).

Sometimes the teachers put software (such as LOGO) on the turtle that allowed you to enter angles and distances, so you could say "rotate 90, move 5, rotate 45, move 6," and so on, but that was for the nursery kids. You used it with the Cartesian (x, y) scale. The deal was you put a piece of paper under the turtle and did stuff like working out how to draw a cube or (even harder) an equilateral triangle. If you could work out the latter, they would even let you attempt to write your name with the turtle.

Turtles were really cool in the early 8-bit computer days. Kids actually ran to math class to use them, and they found out later that the turtle had fooled them into learning Cartesian geometry.

So anyway, what does this have to do with Flash? Well, Flash uses something that's recognizable as turtle graphics. In fact, most vector graphics actually *are* turtle graphics at the most basic level, and thinking in terms of the little turtle trundling around on cheap paper in real time is easier than imagining the Flash graphics engine drawing stuff with scripts and events firing off, running at 12 fps. Indeed, the turtle makes it easy to visualize. Kids were learning the Pythagorean theorem at the age of 5 with these things, so it must be true!

Drawing lines

Flash has a virtual turtle that controls the drawing API. It can move to a point (x1, y1) and then draw to a point (x2, y2). To draw a line from (50, 50) to (100, 100), you do the following:

1. Move to (50, 50).
2. Draw to (100, 100).

Easy, huh? Let's get some practice by working through an example. The following code is included in the download for this chapter as lines01.fla. In a new movie, you'll add a script on frame 1 of the timeline. You don't have to rename the layer or anything. You're going to do everything in ActionScript, and you need only that one layer and one keyframe.

1. Open a new FLA with File ➤ New. Select frame 1 of the timeline and pin the Actions panel to this frame.

   The first thing you need is some paper. Let's create some virtual paper—a movie clip, to be precise.

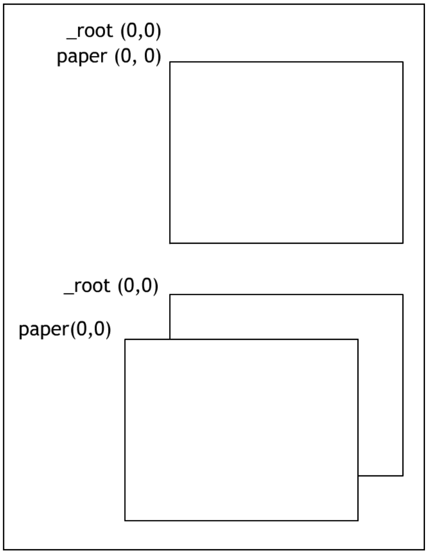
2. In the Script pane, type the following:

   ```
   var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
   ```

   This creates an empty movie clip called paper at depth 100, plus a reference (path) to it also called paper. We chose 100 in case you want to go and add some other stuff onscreen later.

Where is this new movie clip? Well, it's at (0, 0), the top-left corner on the stage, and on timeline _root (because the _x and _y properties of a newly created clip are 0, and the scope of the code is _root, the timeline it's on).

Here's an important point: as long as you leave the clip where it is now, the coordinate space of the main timeline is the same as the local coordinates of the movie clip. As soon as you move the clip, however, the position of a turtle in _root becomes different relative to a turtle on the piece of paper called paper, even if they head for the same point (x, y). Basically, the coordinates of the paper movie clip are wholly independent of the _root coordinates. As shown in the following image, the internal coordinates of paper will show a different origin (0, 0) from that of _root after being moved:



There are times when having the paper in a different place is a good thing, but not while you're coming to grips with the drawing API, so leave it where it is for now.

Don't forget that the y-axis is "down" in Flash, not "up" as it is in math. This is because web design is based around web pages, and they read from top to bottom.

3. Next, you want to define your pen. A turtle could draw different lines depending on which pen you had in it, and the drawing API is no different. You select the line style of the turtle on the movie clip myClip with this:

```
myClip.lineStyle(thickness, color, alpha);
```

The pen will have a thickness of 3, a color of red (0xFF0000), and an alpha value of 100% for the turtle on paper, so type the following line under the script you already have:

```
paper.lineStyle(3, 0xFF0000, 100);
```

4. You now have to move to the start point. The method to do this is moveTo(x, y). So if you want to move to (50, 50), you use this line:

```
paper.moveTo(50, 50);
```

Add this line to the end of what you have so far.

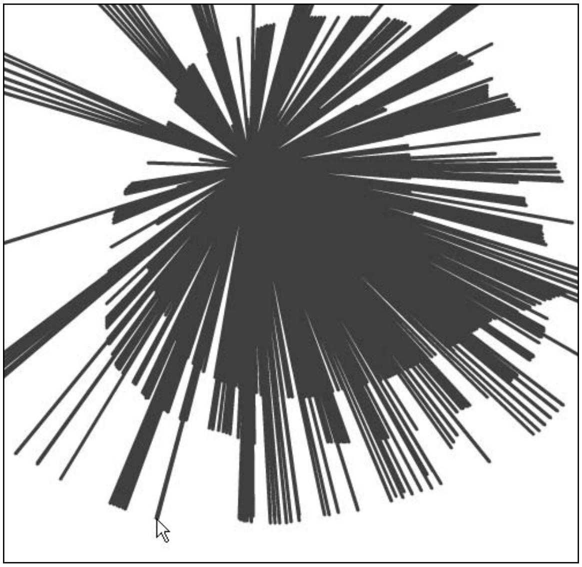
The turtle is now in position at the start of the line. It needs to put the pen down and draw a line as it travels to (100, 100). The method you use to draw is lineTo(). Here's the code so far, plus the new line you need at the end:

```
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
paper.lineStyle(3, 0xFF0000, 100);
paper.moveTo(50, 50);
paper.lineTo(100, 100);
```

On testing this code, you should see a short diagonal line near the top-left corner that points to the bottom-right corner. That's fine, but there's not much happening after the initial elation. Time for a little interactivity. Instead of moveTo() to a fixed distance and then lineTo() to another fixed distance, you'll use lineTo (_xmouse, _ymouse), and you'll do this every time the mouse moves.

5. Change your script to the following (the second line is new, and the moveTo() and lineTo() arguments have changed), which you'll find in the source code folders as lines01.fla:

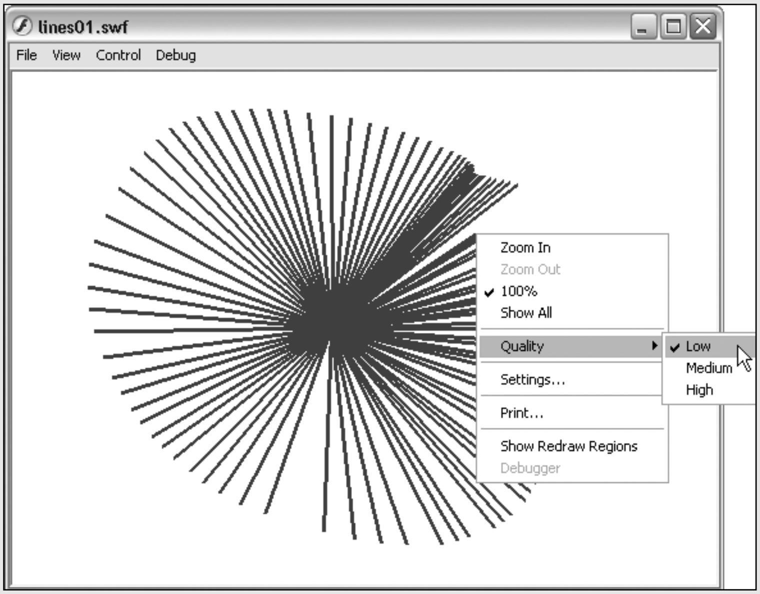
```
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
paper.onMouseMove = function() {
  paper.lineStyle(3, 0xFF0000, 100);
  paper.moveTo(250, 200);
  paper.lineTo(_xmouse, _ymouse);
};
```



Wow, an explosion in a paint factory!

Notice that this script gets slow if you allow a large number of lines to be drawn.

Digression alert! Well, as you might have gathered from the old-school game in the last chapter and various other references in the book, we have a true fondness for the days of the 8-bit home computer. In those times, the screen resolution was minor and graphics were jagged. Lovely! However, if you missed out on those days and want to emulate how graphics looked back then, right/Ctrl-click the movie and select Quality ➤ Low *to switch off anti-aliasing. Besides looking nostalgic and cool, the movie will run faster in this mode because Flash Player doesn't have smooth out the vector strokes.*



A few years ago, the Low *setting was used for style by some Flashers because of its texture and jagged 8-bitness. Although it is used less frequently these days, some Flashers (including James Paterson of* `www.presstube.com`*) still use it because they favor the raw texture and the subsequent speed increase.*

To start a movie in low quality without any user interaction, place the following line of code at the start of your movie:

```
_quality = "low";
```

It will then run in low quality automatically.

Now, where were we? Oh yes. Switch the movie back to High quality and you'll see the movie slow down again because of the number of lines being drawn on the screen at any one time. Although you could probably settle for jagged 8-bit graphics and the associated performance, let's next look at how to fix the performance some other way.

This is one situation where bitmap caching certainly can't help you with performance. As we covered in Chapter 8, any changes to movie clip content require Flash Player to recache the movie clip content. With each new line added here, Flash would have to totally recache this movie clip.

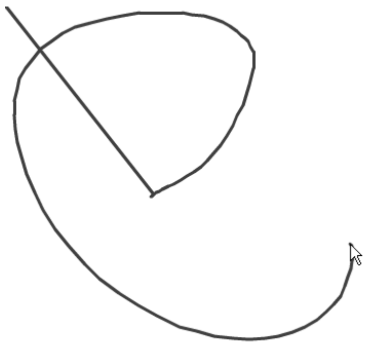
One way of improving performance is by clearing some of the excess lines every now and then. Although you can't *undraw* lines, you can *clear* the whole movie clip with `myClip.clear()`.

Change your script as follows (or open `lines02.fla` from the download files for this chapter):

```
var maxLines:Number = 100;
var lines:Number = 0;
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
paper.onMouseMove = function() {
  lines++;
  if (lines < maxLines) {
    paper.lineStyle(3, 0xFF0000, 100);
    paper.moveTo(250, 200);
    paper.lineTo(_xmouse, _ymouse);
  } else {
    paper.clear();
    lines = 0;
  }
  updateAfterEvent();
};
```

Here you're limiting yourself to a `maxLines` of 100. As long as `lines < maxLines`, you do the original `moveTo()`/`lineTo()` actions in the clip `paper`. As soon as 100 lines have been created, you clear the `paper` movie clip with `paper.clear()`and reset `lines` back to 0. You also add `updateAfterEvent()` at the end of the `onMouseMove` to make Flash redraw the stage on every mouse movement, rather than on every frame. This should make the animation much smoother.

6. You can also make your lines into one continuous line simply by commenting out or removing the `moveTo()` action. You'd then see something like this:



If you can't visualize what's happening here, think back to the turtle. Rather than going back to the center of the screen after every mouse move, the turtle just waits for you to move the mouse, and then it goes to the last position you were *from where it is now*. It's effectively following you. If it does this fast enough, all your straight line segments get short and therefore start to look like curves.

You have the beginnings of a drawing tool here, which you might expand upon in time. Hopefully by the end of this chapter, you'll be aware of all the drawing options available through Flash's drawing API and it will be within your grasp.

The example file, `record_drawing.fla`, available in the chapter download, shows how drawing data can be stored in an array for reproduction. Although this isn't the most processor-friendly way of storing such information, it begins to illustrate the concept nicely.

Enough straight lines for now. Let's add curves to your drawing repertoire.

Drawing curves

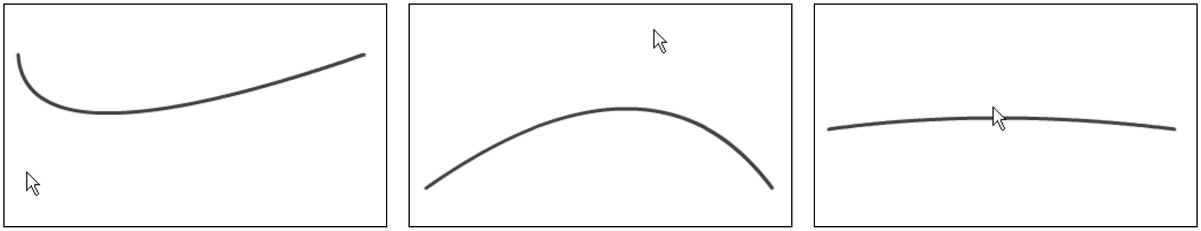
The curve functionality of the drawing API is best understood through a demonstration. It's time for another script! Add the following script to the first frame of a new movie. You're again going to use ActionScript only, so you need just the script and a single frame.

Create a new movie with File ➤ New ➤ Flash Document, and select and pin frame 1 of the only layer. Add the following script in the Script pane:

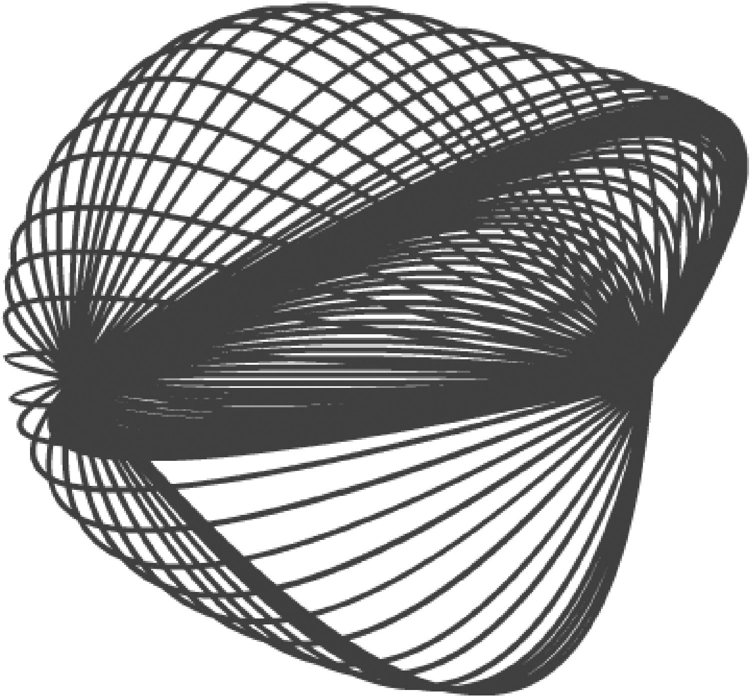
```
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
paper.onMouseMove = function() {
  paper.clear();
  paper.lineStyle(3, 0xFF0000, 100);
  paper.moveTo(100, 200);
  paper.curveTo(_xmouse, _ymouse, 450, 200);
  updateAfterEvent();
};
```

You can find this script in the file `curves01.fla`.

Note that you're now using a new method, `curveTo()`, in place of `lineTo()`. You're moving the turtle to point (100, 200), center-left of the stage, and then drawing to (450, 200), center-right of the stage. But something happens to the turtle in between. It's almost as if the mouse was some sort of turtle magnet. The turtle tries to get to point (450, 200), but it's attracted to the "magnet" and gets pulled away toward the mouse. The turtle's new path is a *curve*:



Notice that you're also clearing the paper every time. This means you delete the previous line and draw a new line every frame, and the resulting curve looks animated. Comment out the `paper.clear()` *command to see all the individual curves that make up the animation. Isn't that pretty? We have a feeling that lots of geometrical screen savers and "Please wait . . . loading" eye-candy screens are in the cards!*



Looking at the difference between the `lineTo()` and `curveTo()` arguments, you can see what's going on:

```
moveTo(100, 200);
lineTo(450, 200);
moveTo(100, 200);
curveTo(_xmouse, _ymouse, 450, 200);
```

The first two lines of code cause a line to be drawn. The turtle moves to (100, 200) and then draws to (450, 200). The second pair of lines contains the `curveTo()` action, which has the (450, 200) point specified but also has a new set of control points (_xmouse, _ymouse). That explains why the mouse is doing stuff to the curve: its position is being used somehow.

So you have

```
curveTo(controlPointX, controlPointY, x, y);
```

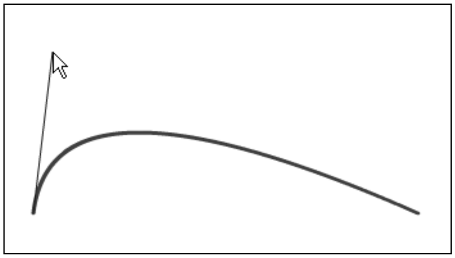
curveTo() is just like lineTo() in that it causes the turtle to go to (x, y) with the pen down, drawing a line as it goes. Unlike lineTo(), however, curveTo() has a control point made up of (*controlPointX*, *controlPointY*) that affects the path the turtle takes. The turtle seems to want to go through the control point, but it doesn't do it, so it *tends toward* the control point. OK, that's fine, but you really need more information than that to know that the control point is well enough to actually create a curve you can predict.

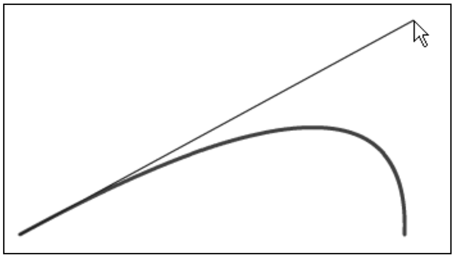
There are two ways to explain this. We can discuss quadratic Bezier curves and stuff like that for a couple of pages (and probably send half of you to sleep in the process), or we can tell you what the curveTo() method actually does in nonmathematical words and pictures. The pretty pictures option wins hands down every time for us. Let's look at a quick example.

Create a new movie and attach this script to frame 1 of the main timeline (or just open the file curves02.fla):

```
var red:Number = 0xFF0000;
var blue:Number = 0x0000FF;
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
paper.onMouseMove = function() {
  paper.clear();
  // draw curve...
  paper.lineStyle(3, red, 100);
  paper.moveTo(100, 200);
  paper.curveTo(_xmouse, _ymouse, 450, 200);
  // draw control point tangent...
  paper.lineStyle(0, blue, 100);
  paper.moveTo(100, 200);
  paper.lineTo(_xmouse, _ymouse);
  updateAfterEvent();
};
```

This gives you the picture you want, and it's interactive as well. The thin line is the control line. It's a line joining the start point to the control point, and the curve forms itself so that it's *tangential* to the control line:





Also worth looking at is the file drawAPI.fla. This FLA contains fully interactive versions of a line and curve to show you what the lineTo() and curveTo() methods are capable of. Click any control point and you'll find you can drag it to change the shape of the attached line or curve. Adventurous readers may also want to look at the code, which is written in modular motion graphics code and contains a rather nifty trick for drawing circles with a single line. See if you can work it out!

You now have everything you need to construct a simple interactive drawing tool. Create a new movie yet again and attach the following script to the first frame of the main timeline, or just open the file elasticBand.fla:

```
function spaceBarPress():Void{
  endX = end2X = _xmouse;
  endY = end2Y = _ymouse;
  penToggle = false;
}
function penHandler():Void{
  this.clear();
  if (!penToggle) {
    // if penToggle is false, use line pen
    this.lineStyle(0, blue, 100);
    this.moveTo(endX, endY);
    this.lineTo(_xmouse, _ymouse);
  } else {
    // penToggle must be true,
    // so use curve pen
    this.lineStyle(0, red, 100);
    this.moveTo(endX, endY);
    this.curveTo(_xmouse, _ymouse, end2X, end2Y);
  }
  updateAfterEvent();
}
function paperHandler ():Void{
  if (!penToggle) {
    // first press...
    // toggle to curve pen
    // and set second endpoints of curve
    penToggle = true;
    end2X = _xmouse;
    end2Y = _ymouse;
  } else {
    // second press...
    // Draw the curve to the paper...
    this.lineStyle(0, black, 100);
    this.moveTo(endX, endY);
    this.curveTo(_xmouse, _ymouse, end2X, end2Y);
    // reset pen back to line pen...
    penToggle = false;
    endX = end2X;
    endY = end2Y;
  }
};
//
```

```
// Initialize Objects & Variables...
var keyboard:Object = new Object();
var endX:Number = _xmouse;
var end2X:Number = _xmouse;
var endY:Number = _ymouse;
var end2Y:Number = _ymouse;
var penToggle:Boolean = false;
var red:Number = 0xFF0000;
var blue:Number = 0x0000FF;
var black:Number = 0x000000;
// Create clips...
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
var pen:MovieClip = this.createEmptyMovieClip("pen", 101);
//
// define handlers...
keyboard.onKeyDown = spaceBarPress;
Key.addListener(keyboard);
pen.onMouseMove = penHandler;
paper.onMouseDown = paperHandler;
```

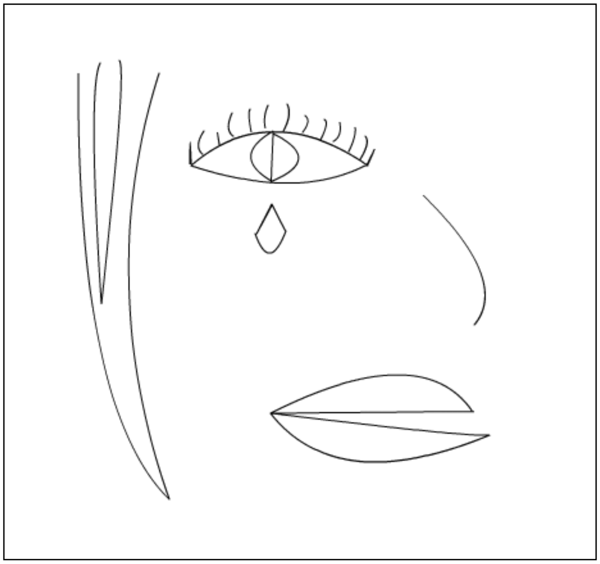
There are two movie clips: paper and pen. The pen clip is where the cursor lines sit—that is, where curves and lines in progress are drawn. When you're ready to fix a curve, it's transferred to the movie clip paper. You have to do this because to animate your lines in progress (your "pen"), you have to constantly clear the pen movie clip. This means that you can't draw anything permanent in it as well, so the movie clip paper is where you place your lines once you want them to be indelible.

To draw a line, you move the blue line around to the first point to which you want to draw. Click when you have your mouse at the right place.

> *Notice that if you split the paper into several movie clips—*paper01*,* paper02*,* paper03*, and so on—you'll have the beginnings of drawing layers right away, with no fuss.*

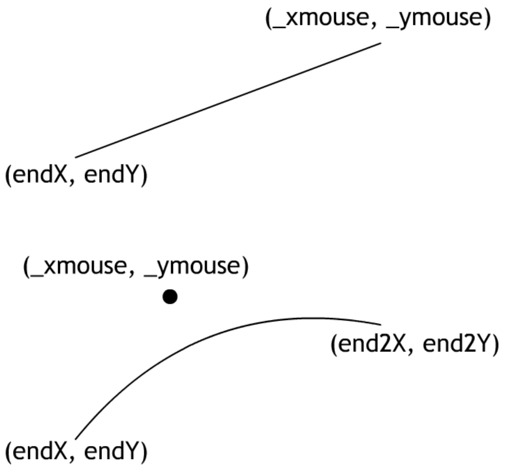
The line will then turn red. If you click again without moving the mouse, you'll get a permanent black line. If you want to draw a curve, move the cursor while the line is red to form a curve. Once you're happy, simply click again and the red curve will become a permanent black curve. So, double-click to make a straight line, single-click to bend the line, and single-click again to make a curve.

Also, to move your cursor without drawing, just press and hold any key while you move. By doing this you can easily make a simple drawing consisting of lines and curves. Not quite enough to make Picasso jealous, but pretty neat anyway.



How the sketching code works

As it stands, the basic code works as follows: all lines are drawn from point (endX, endY) to either the mouse position (_xmouse, _ymouse) for a line or to (end2X, end2y) with the mouse position as the control point for the curve:



When you draw the current line style permanently to paper, the pen has two states: you've clicked either once (so the pen draws a blue line) or twice (so the pen draws a red curve). This is controlled by the variable penToggle.

Filling shapes

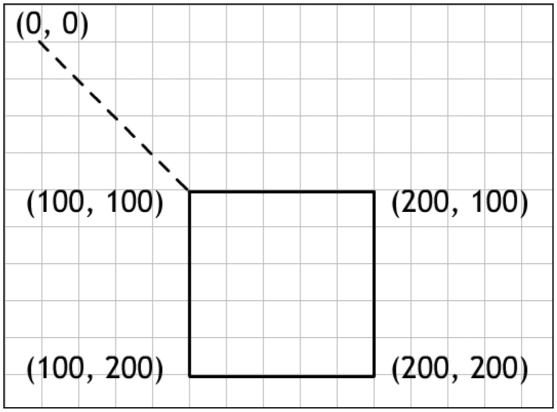
OK, you're moving on to the final bit of the drawing API: **fills**. Obviously, the poor virtual Flash turtle can draw only lines, so how do you visualize it drawing a fill? The answer is that it trundles around the *perimeter* of the area you want to fill and then politely asks the drawing API to fill the area it just defined.

This perimeter is called the **fill path**, or just **path**. Although the detail of the fill part of the drawing API is a little hairy, the fundamental principle is easy. As usual, an example will help to show you how to use it.

1. First you create a movie clip object to draw into. Create a new Flash document and attach the following script to frame 1:

```
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
```

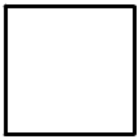
2. Now that you have something to draw into, you want to draw around the edge of your shape. Here, you'll draw the following square:



You have to move from the origin (0, 0) and draw out your square path. To do this, add the following lines:

```
paper.lineStyle(3, 0x000000, 100);
paper.moveTo(100, 100);
paper.lineTo(200, 100);
paper.lineTo(200, 200);
paper.lineTo(100, 200);
paper.lineTo(100, 100);
```

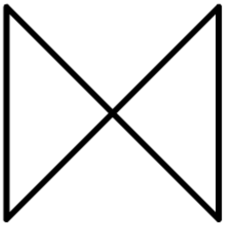
You've given the turtle a pen of thickness 3, a color of black (0x000000), and an alpha of 100%. The turtle then moves from the start position to (100, 100) and proceeds to draw around the perimeter in a clockwise direction. If you run the movie, you'll see the finished path drawn.



An important point to notice is that you must get the order of the corner points that you make the turtle visit correct. If the points are out of sequence, the turtle will draw incorrectly, as the following script demonstrates:

```
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
paper.lineStyle(3, 0x000000, 100);
paper.moveTo(100, 100);
paper.lineTo(200, 200);
paper.lineTo(200, 100);
paper.lineTo(100, 200);
paper.lineTo(100, 100);
```

This looks like an hourglass or a rotated bow tie—clearly not what you want! Flash would still fill the path if you asked it to, but it would fill out the two enclosed triangles rather than the square you want filled.



3. To fill the square, you first need to define the fill color. You do this via the beginFill() method.

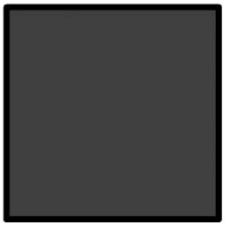
   ```
   myClip.beginFill(color, alpha);
   ```

   color refers to the color of the fill, and you can also set alpha. beginFill() tells Flash one other important thing. Essentially it says to the drawing API "I'm setting my turtle off on the path I want you to fill." To tell Flash that the turtle is done, you simply include an endFill(x, y).

4. Change the code as highlighted here:

   ```
   var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
   paper.lineStyle(3, 0x000000, 100);
   paper.beginFill(0xFF0000, 100);
   paper.moveTo(100, 100);
   paper.lineTo(200, 100);
   paper.lineTo(200, 200);
   paper.lineTo(100, 200);
   paper.endFill(100, 100);
   ```

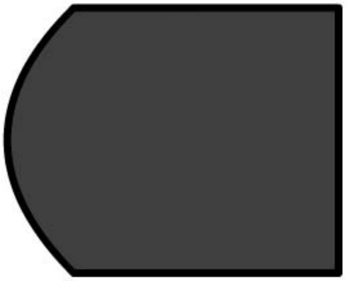
   The first and second lines create a movie clip to draw into and set the line style. Line 3 tells Flash you've started defining your path, and it also tells Flash to use a red fill (0xFF0000) when the path is completed. Lines 4 through 7 define the area to be filled. The last lineTo() is now replaced by an endFill(). This tells Flash to draw to the point (100, 100) and to fill the path defined by the turtle since the initial beginFill(). You'll see the red square with black border as shown in the following image if you run this FLA (or open fillSquare.fla):



You can also use curves instead of lines. Add the following line where highlighted:

```
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
paper.lineStyle(3, 0x000000, 100);
paper.beginFill(0xFF0000, 100);
paper.moveTo(100, 100);
paper.lineTo(200, 100);
paper.lineTo(200, 200);
paper.lineTo(100, 200);
paper.curveTo(50, 150, 100, 100);
paper.endFill(100, 100);
```

This code creates the following shape:



Notice that because you have to end your path definition with an endFill(), and the line you want to end with is now a curveTo(), you're a little stuck with the endFill() since there are no more lines to create in the path. Instead, you simply endFill() to the point you're already at, (100, 100).

You can, of course, animate the filled shapes using clear() and redraw a transformed version of the same shape every frame. You'll find the following code within the file fillCartoon.fla:

```
var paper:MovieClip = this.createEmptyMovieClip("paper", 100);
var vertexX:Array = new Array(100, 200, 200, 100);
var vertexY:Array = new Array(100, 100, 200, 200);
paper.onEnterFrame = function() {
  for (var i:Number = 0; i<4; i++) {
    vertexX[i] += (Math.random()*10)-5;
    vertexY[i] += (Math.random()*10)-5;
  }
  paper.clear();
  paper.lineStyle(3, 0x000000, 100);
  paper.beginFill(0xFF0000, 100);
  paper.moveTo(vertexX[0], vertexY[0]);
  paper.lineTo(vertexX[1], vertexY[1]);
  paper.lineTo(vertexX[2], vertexY[2]);
  paper.lineTo(vertexX[3], vertexY[3]);
  paper.endFill(vertexX[0], vertexY[0]);
};
```

This time you assign the four points of your square to a couple of arrays. The points are now (vertexX[0], vertexY[0]) to (vertexX[3], vertexY[3]), and you're varying their positions by a small random value every onEnterFrame (try different random increments for vertexX and vertexY, and different frame rates to find a nice organic animation).

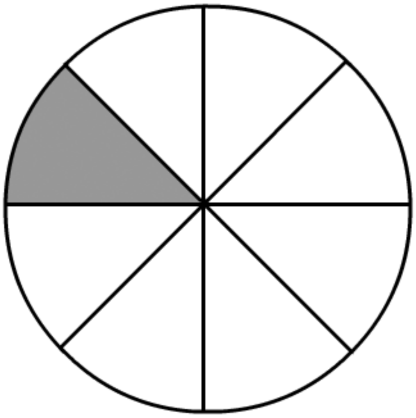
Flash also allows you to define gradient fills. We recommend that you don't do this for dynamic animation until you're fully up to speed with the solid color fills. Flash uses a number of objects to define the gradient and, as with gradient fills in general, it's a little slow if you try to animate your shapes as you've just done with solid fills.

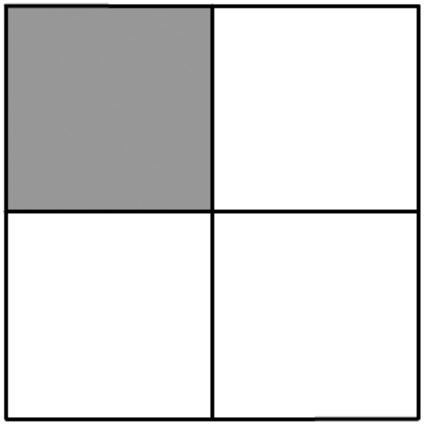
Now that you have an idea of how the drawing API works, you'll get your hands dirty and start playing. One of the cool things about the drawing API is that you can build a generalized bit of code to draw something (the **draw engine**), and then just go ahead and start playing around and experimenting. That's what the experts do in the second edition of the *Flash Math Creativity* book (friends of ED, 2005). You'll do something similar, although you'll keep the math rather closer to the ground, so don't worry!

First, though, you need a starting point.

Creating a kaleidoscope

Perhaps you've seen a toy consisting of a little kaleidoscope on a string. You can twist it around and it creates lots of pretty patterns. It's all done with mirrors, of course. In a kaleidoscope, only one segment (the grayed-out one in the following diagrams) actually contains the colored beads. The others are just reflections of that segment. We thought maybe we could do something similar with Flash, although to keep the math simpler we'll base it on quadrants rather than the kaleidoscope's eightfold symmetry.

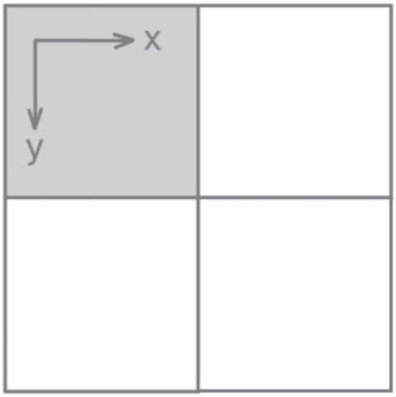




Let's go with this and have a look at the math.

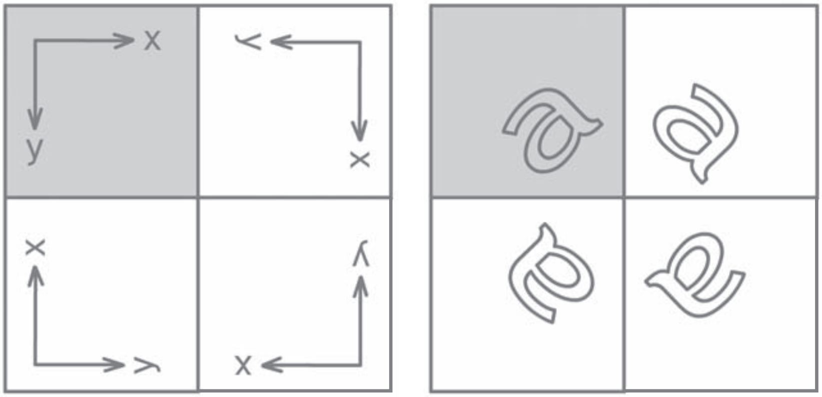
Kaleidoscope math

To set up a kaleidoscope, you first need to understand how a real kaleidoscope works. A kaleidoscope works via *reflections*. In math-speak, a reflection is simply a *copy*. There's a difference between the original and the reflections in a real kaleidoscope: the reflections are *rotated*. You rotate the original segment to create copies that form a circle. To achieve this effect in Flash, you need to perform a similar operation.



So, if the original segment has the orientation and position shown in the preceding image, you need to *rotate* the other three copies. Generally, you need to rotate each segment by 360 degrees divided by the total number of segments. So if you have four segments, you need to rotate the first copy by 90 degrees, the second by 180 degrees, and the third by 270 degrees.

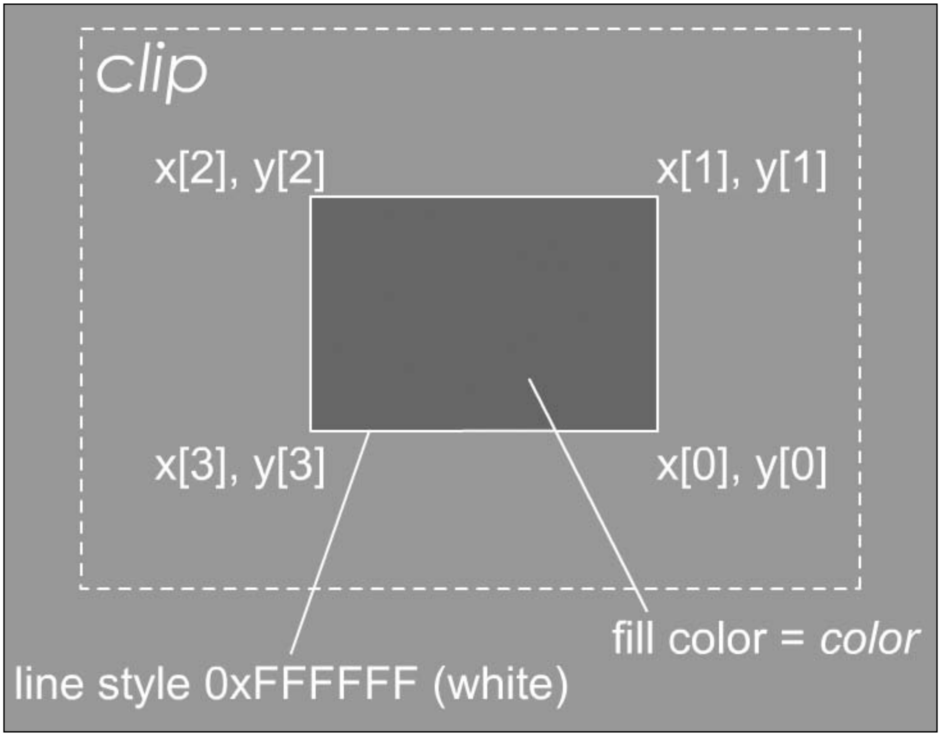
The next step is to put all that into code.



Building the kaleidoscope engine

You'll start off by drawing a simple shape (a rectangle) in a movie clip. To do that, you need to select a line style and a fill color, and then move between the corner points within a `beginFill()...endFill()` set of actions. To define the corner points, it would be wise to hold them in an array, because you can then easily add more corner points later if you want to extend the code.

This function will draw out a general rectangle as shown in the following image. It will draw it within a movie clip specified by the argument `clip`, with a white border and a fill color defined by the argument `color`.



```
function drawQuad(clip:MovieClip, color:Number):Void {
  clip.lineStyle(1, 0xFFFFFF, 0);
  clip.beginFill(color, 100);
  clip.moveTo(x[0], y[0]);
  clip.lineTo(x[1], y[1]);
  clip.lineTo(x[2], y[2]);
  clip.lineTo(x[3], y[3]);
  clip.endFill(x[0], y[0]);
}
```

It's not obvious how to create a rectangle without a border. To do this, you can set a line style with an alpha of 0, but this means that Flash will still draw a border (it's just that you can't see it). That's cool, but it means that it slows Flash down, and when you get to the heady heights of creating a 3D vector engine, you'll want all the performance you can squeeze out of Flash Player. A better but somewhat obscure way to create a rectangle so that Flash doesn't even draw the lines is to set a line thickness of `undefined`. If you replace the following line:

```
clip.lineStyle(1, 0xFFFFFF, 0);
```

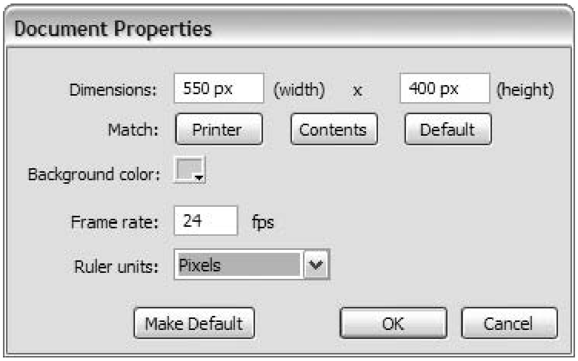
with this line:

```
clip.lineStyle(undefined, 0, 0);
```

Flash won't draw the lines at all, and your code will run faster because of it.

Let's use this function in a bit of code.

1. Create a new FLA. Select Modify ➤ Document and set the background color to a light gray (#CCCCCC) and the frame rate to 24 fps.



Document Properties

Dimensions: 550 px (width) x 400 px (height)

Background color:

Frame rate: 24 fps

Ruler units: Pixels

2. Select frame 1 of the only layer and enter the following code in the Script pane (or have a look at `pattern01.fla`):

```
function drawQuad(clip:MovieClip, color:Number):Void {
  clip.lineStyle(undefined, 0, 0);
  clip.beginFill(color, 100);
  clip.moveTo(x[0], y[0]);
  clip.lineTo(x[1], y[1]);
  clip.lineTo(x[2], y[2]);
  clip.lineTo(x[3], y[3]);
  clip.endFill(x[0], y[0]);
}
//
// Define colors
var color1:Number = 0xDD8000;
var color2:Number = 0xAACC00;
var color3:Number = 0xFFF287;
var color4:Number = 0x6699FF;
// Define screen extents for later use
Stage.scaleMode = "exactFit";
var middleX:Number = Stage.width/2;
var middleY:Number = Stage.height/2;
Stage.scaleMode = "noScale";
// create clip...
var quad1:MovieClip = this.createEmptyMovieClip("quad1", 1);
quad1._x = middleX;
quad1._y = middleY;
// Initialize points...
var width:Number = 100;
var height:Number = 100;
var x:Array = new Array();
var y:Array = new Array();
x[0] = 0;
y[0] = 0;
x[1] = 0;
y[1] = -height;
x[2] = -width;
y[2] = -height;
x[3] = -width;
y[3] = 0;
drawQuad(quad1, color1);
```

The additional code will create an empty movie clip called quad1 and place it in the center of the screen. It will then draw out a rectangle with proportions defined by the height and width variables.

OK, that's one static square. You're in the game of creating motion graphics, though, so how do you make the drawing API create animated content? Well, you can clear all the content in a movie clip and redraw it. Change the function to include the clear() line as shown here (you can see the full effect for this in pattern02.fla):

```
function drawQuad(clip:MovieClip, color:Number):Void {
  clip.clear();
  clip.lineStyle(undefined, 0, 0);
  clip.beginFill(color, 100);
  clip.moveTo(x[0], y[0]);
  clip.lineTo(x[1], y[1]);
  clip.lineTo(x[2], y[2]);
  clip.lineTo(x[3], y[3]);
  clip.endFill(x[0], y[0]);
}
```

Also, delete this line from the end of the listing:

```
drawQuad(quad1, color1);
```

and replace it with this:

```
this.onEnterFrame = function() {
  for (var i:Number = 0; i<4; i++) {
    x[i] += Math.random()*4-2;
    y[i] += Math.random()*4-2;
  }
  drawQuad(quad1, color1);
};
```

So what does this new code do? It draws a new rectangle every frame, changing the position of the corner points every time and producing a “wobbly” square.

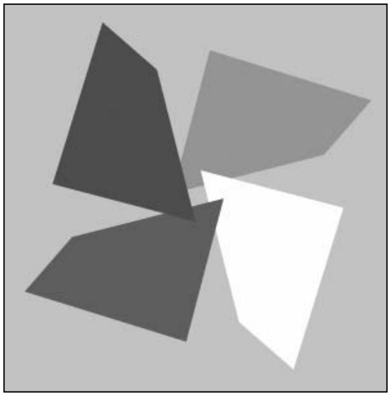
To create the kaleidoscope pattern, you need to make another three rotated versions of quad1. That’s actually very easy. Change the “create clip” section of the code (starting at line 22 of the code), which currently creates only one movie clip (quad1) as follows:

```
// create clip...
var quad1:MovieClip = this.createEmptyMovieClip("quad1", 1);
quad1._x = middleX;
quad1._y = middleY;
var quad2:MovieClip = this.createEmptyMovieClip("quad2", 2);
quad2._x = middleX;
quad2._y = middleY;
quad2._rotation = 90;
var quad3:MovieClip = this.createEmptyMovieClip("quad3", 3);
quad3._x = middleX;
quad3._y = middleY;
quad3._rotation = 180;
var quad4:MovieClip = this.createEmptyMovieClip("quad4", 4);
quad4._x = middleX;
quad4._y = middleY;
quad4._rotation = 270;
```

You also want to draw content into the new clips, so amend the onEnterFrame code where highlighted:

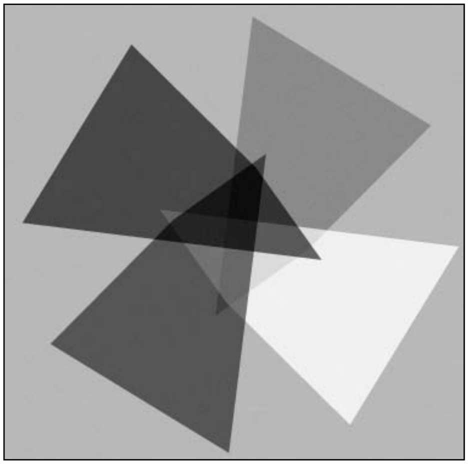
```
this.onEnterFrame = function() {
  for (var i:Number = 0; i<4; i++) {
    x[i] += Math.random()*4-2;
    y[i] += Math.random()*4-2;
  }
  drawQuad(quad1, color1);
  drawQuad(quad2, color2);
  drawQuad(quad3, color3);
  drawQuad(quad4, color4);
};
```

Instead of drawing in just quad1, you're now drawing in all four quadrants, using different colors.



By changing the alpha values of the (now skewed) rectangles, you're drawing via the function drawQuad(), and you can create some pretty overlapping patterns.

```
function drawQuad(clip:MovieClip, color:Number):Void {
  clip.clear();
  clip.lineStyle(undefined, 0, 0);
  clip.beginFill(color, 50);
  clip.moveTo(x[0], y[0]);
  clip.lineTo(x[1], y[1]);
  clip.lineTo(x[2], y[2]);
  clip.lineTo(x[3], y[3]);
  clip.endFill(x[0], y[0]);
}
```

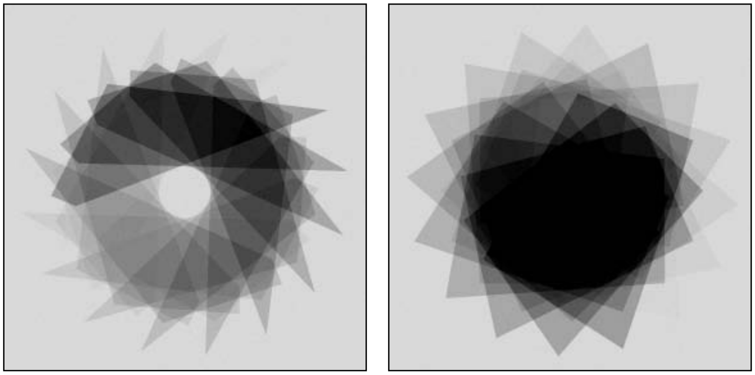


Have a look at pattern04.fla, which develops this train of thought further, except this time you create a separate speed value for each point on the quad, which will “bounce” around within a 200 × 200 square rather than change direction randomly.

```
function drawQuad(clip:MovieClip, color:Number):Void {
  clip.clear();
  clip.lineStyle(undefined, 0, 0);
  clip.beginFill(color, 50);
  clip.moveTo(x[0], y[0]);
  clip.lineTo(x[1], y[1]);
  clip.lineTo(x[2], y[2]);
  clip.lineTo(x[3], y[3]);
  clip.endFill(x[0], y[0]);
}
//
// Define colors
var color1:Number = 0xDD8000;
var color2:Number = 0xAACC00;
var color3:Number = 0xFFF287;
var color4:Number = 0x6699FF;
// Define screen extents for later use
Stage.scaleMode = "exactFit";
var middleX:Number = Stage.width/2;
var middleY:Number = Stage.height/2;
Stage.scaleMode = "noScale";
// create clip...
var quad1:MovieClip = this.createEmptyMovieClip("quad1", 1);
quad1._x = middleX;
quad1._y = middleY;
var quad2:MovieClip = this.createEmptyMovieClip("quad2", 2);
quad2._x = middleX;
quad2._y = middleY;
quad2._rotation = 90;
var quad3:MovieClip = this.createEmptyMovieClip("quad3", 3);
quad3._x = middleX;
quad3._y = middleY;
quad3._rotation = 180;
var quad4:MovieClip = this.createEmptyMovieClip("quad4", 4);
quad4._x = middleX;
quad4._y = middleY;
quad4._rotation = 270;
// Initialize points...
var width:Number = 100;
var height:Number = 100;
var x:Array = new Array();
var y:Array = new Array();
var sX:Array = new Array();
var sY:Array = new Array();
x[0] = 0;
```

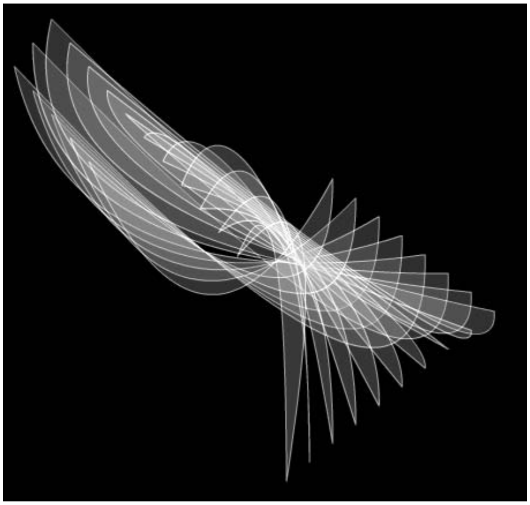
```
y[0] = 0;
x[1] = 0;
y[1] = -height;
x[2] = -width;
y[2] = -height;
x[3] = -width;
y[3] = 0;
for (var i:Number = 0; i<4; i++) {
  sX[i] = (Math.random()*4)+4;
  sY[i] = (Math.random()*4)+4;
}
this.onEnterFrame = function() {
  for (var i:Number = 0; i<4; i++) {
    x[i] += sX[i];
    y[i] += sY[i];
    if (Math.abs(x[i])>200) {
        sX[i] = -sX[i];
    }
    if (Math.abs(y[i])>200) {
        sY[i] = -sY[i];
    }
  }
  drawQuad(quad1, color1);
  drawQuad(quad2, color2);
  drawQuad(quad3, color3);
  drawQuad(quad4, color4);
};
```

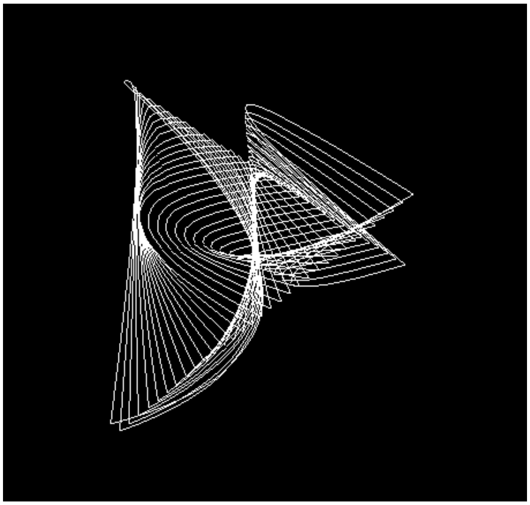
You now have something that has its moments, but the effect is simple. You can generalize the effect to cater to any number of shapes instead of the current four, and this has been done for pattern05.fla. Here, the number of shapes (or “petals,” as the pattern now looks like a flower) is defined by the value sectors, and loops are used to produce the required number of clips.



Groovy!

The final two files in the download for this section start to move away from the concept of the kaleidoscope (but they use what is essentially the same code) to create a totally new effect that users of the Windows platform may recognize as similar to the "mystify" screen savers that ship with their operating system. There are two versions, **flashMystify**, which uses filled curved shapes, and **flashMystifyRetro**, for those who enjoy the old-school stuff, complete with simple vectors (presented in low quality to enhance the effect further):





This sort of experimentation with Flash to create new effects is one way you can create code that you may one day repurpose as part of a new site's navigation. By making such experimentation open source, designers like Joshua Davis made a name for themselves early in Flash's history.

More to the point, you will have noticed that, although this stuff has nothing to do with web design, it's a good way of practicing ActionScript. Also, you'll forget you're using advanced code techniques because the whole thing is so much *fun*!

That's not to say that the drawing API isn't useful. It's a great way to create some nifty UIs. If you look at the Futuremedia site, you'll see that the expanding rectangle effect could have been created using the drawing API, so rather than all that painstaking hand-drawing you did at the beginning of the project, you could have made Flash draw the tricolor from scratch at runtime. A big advantage of doing this is that the code would have a much lower file size than the hand-drawn graphics in the final SWF. The disadvantage is that you would have had to learn the drawing API at the beginning of the book, and given that it's an advanced technique, it wasn't a very good idea! There's nothing stopping you from doing some stuff with the drawing API once you have a finished Futuremedia site at the end of the book, though.

Book project: Color transition event handling and data

In this section, you'll complete the event-driven site animations in the Futuremedia site by adding the final part of the transition that occurs when you click a strip: the color fades. You'll then start to think about adding some data to allow Flash to create the site structure.

So far you have the position transition sorted out, but the color transition isn't implemented. At the moment, when you click a strip, the selected strip zooms in correctly, but it doesn't split off into three new strips to present further navigation options to take you deeper into the site.

Wiring the colTransition() function

The starting point of this section is your work-in-progress file. You can also use the file `futuremedia_code02.fla` if you want to double-check your work or play it safe.

At the moment, the function `colTransition()` is neither complete nor linked up to the previous events, `navigate()` and `posTransition()`. Let's link `posTransition()` up to `colTransition()` first.

At the moment, the latter part of the `posTransition()` function (from line 25 onward) looks like this (the highlighted section will be changed in a moment):

```
            delete this.onEnterFrame;
            // then place tricolor back at the start position…
            tricolor._y = BORDER;
            tricolor._xscale=tricolor._yscale=100;
            selectedStrip = this._name.substring(5, 6);
            selectedCol = _root["color"+selectedStrip];
            tricolor.setCol(selectedCol, selectedCol, selectedCol);
        }
    };
```

You need to wire up the colTransition() function to your animation sequence so far. At the end of the position transition code (function posTransition()), you'll extend the animation to include your (as yet unwritten) function stub colTransition(). To do this, you need to change the following line in posTransition:

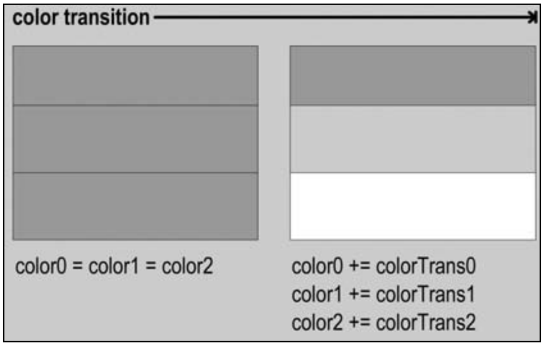
```
delete this.onEnterFrame;
```

to

```
this.onEnterFrame = colTransition;
```

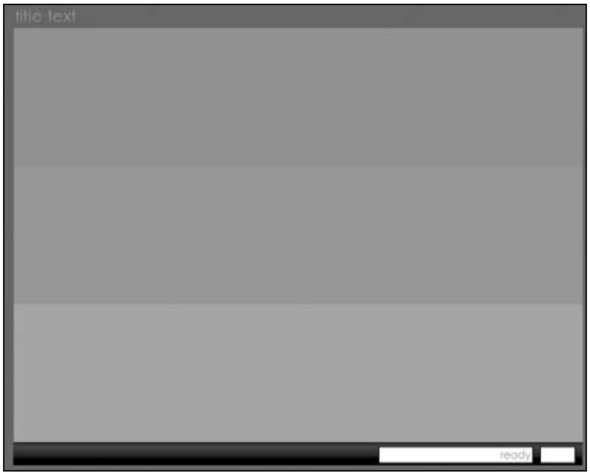
The game plan

So what do you want to do? You guessed it, it's storyboard time! Everyone sitting comfortably? Let's begin . . .



At the end of the position transition, which is the point you are in (looking again at futuremedia_code02.fla) after you click any strip, all three strips are the same color. If you click the top strip, you'll end up with an orange rectangle. In this case, color0 = color1 = color2 = orange (or, in numbers, 0xDD8000).

You need to fade the three strip colors to three new different colors that are based on the *original* color of the strip you clicked (in this case, shades of orange).



You'll do this by making all three final colors brighter, which you can achieve by adding something to the current color value. The preceding storyboard shows the addition of the three values `colorTrans0`, `colTrans1`, and `colorTrans2`. So the `colTransition()` code has to fade from the three monocolored strips to the three uniquely colored strips.

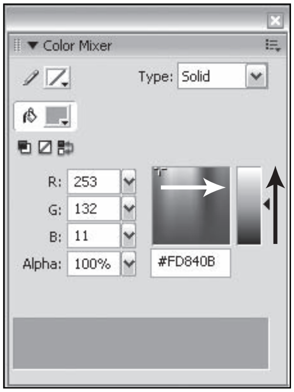
OK, that's the direction you'll take. You now need to know how to implement it. You need to ask yourself the following two questions:

- How do I perform the fade so that it's smooth?
- What will my target colors (`color + colorTrans`) be?

Fading color

If you want to fade between black (0x000000) and white (0xFFFFFF) in a gradual transition, how would you do it? Well, you could simply add 1 to the starting color every frame until you reach the final target color, white. That doesn't actually work. You don't get a smooth transition from black to white through all the grays; you get a transition from black to white that goes through all the colors of the rainbow.

The way to do it is similar to the way you mix colors in the Color Mixer panel when you create new colors with the RGB model: you treat the red, green, and blue components *separately*. So to move from black to white while only sticking to the colors that are actually between black and white (i.e., all the grays) rather than different hues (red, blue, green, and so on), you have to add 1 to the red component, 1 to the green component, and 1 to the blue component every time. You can see this effect if you try it manually in the Color Mixer panel.



Type: Solid
R: 253
G: 132
B: 11
Alpha: 100%
#FD840B

If you click the crosshairs in the color swatch and drag it in the direction of the white arrow, you'll see that the color changes through the colors of the rainbow (or the colors of the visible spectrum for the physicists in the house). As you do this, you'll see that the R, G, and B values change in sequence, and the hex color value at the bottom left goes up linearly. That isn't what you want. You want a more subtle effect, where color is maintained and the only thing that changes is brightness. You can do this if you click-drag the little left-facing arrow to the right of the color swatch upward in the direction of the black arrow. You'll see that the R, G, and B values all go up at the same rate, and *all* the digits of the hex value seem to change at the same time. This is closer to the effect you want.

Red tends to add "warmth" to an RGB color. Keeping this value the same will make for pages that seem to have a consistent "mood." So what you really need to do is change the green and blue channels without varying the red channel.

> *Although we talk about fading color, the code actually goes toward brighter colors, so you're creating "fade to white" rather than the usual "fade to black."*

Coding the color transition

Let's start coding. You first need to define your color variables.

1. Add the following code to the INITIALIZE section (around line 82):

   ```
   // Define initial screen colors
   // (further colors will use these colors as their seed)
   var color0:Number = 0xDD8000;
   var color1:Number = 0xAACC00;
   var color2:Number = 0x0080CC;
   // Define transition colors
   var colTrans:Array = new Array(0x000880, 0x001100, 0x002280);
   var fadeColor:Number = 0;
   // Define navigation strip variables...
   ```

   This defines your target colors when you want to recolor your strip; color0's target becomes color0+colTrans[0], color1's target is color1+_colTrans[1], and so on. Notice that color0 is the closest to the original color, and color2 is the furthest away (because you're adding a bigger value to it). You'll also use a variable, fadeColor, that will allow you to slowly add the transition value over time, resulting in a fade effect.

2. Next up, you'll add lines to the functions posTransition() and colTransition() to use the new variables. Add the following line to the end of posTransition(). This sets fadeColor to 0, so you reset the fade effect at the start of every new transition.

   ```
       selectedStrip = this._name.substring(5, 6);
       selectedCol = _root["color"+selectedStrip];
       tricolor.setCol(selectedCol, selectedCol, selectedCol);
       fadeColor = 0;
     }
   }
   ```

3. Add the following code to completely replace the existing colTransition() function:

```
function colTransition():Void {
  fadeColor += 0x000202;
    if (fadeColor < colTrans[0]) {
      tricolor.colstrip0.setRGB(selectedCol+fadeColor);
    }
    if (fadeColor < colTrans[1]) {
      tricolor.colStrip1.setRGB(selectedCol+fadeColor);
    }
    if (fadeColor < colTrans[2]) {
      tricolor.colStrip2.setRGB(selectedCol+fadeColor);
    } else {
      // we are now at the new page...
      color0 = tricolor.colStrip0.getRGB();
      color1 = tricolor.colStrip1.getRGB();
      color2 = tricolor.colStrip2.getRGB();
      delete this.onEnterFrame;
    }
}
```

So what does this code do? Well, it slowly increments fadeColor, which increases from 0x000000. This means fadeColor starts off as black and slowly gets brighter. You add this color to the starting colors, color0, color1, and color2, and using the color objects colStrip0 through colStrip2 you defined earlier via the function stripCols (and which are inside tricolor), you gradually add fadeColor to the existing strip colors. You stop adding this new color to each strip once it has reached its target color, colTrans.

Because strip 2 is the one with the highest colTrans value, you know all three color transitions are complete when this one is done, so that's when you stop. When you've completed the full color change, you replace the original colors with the new strip colors:

```
color0 = tricolor.colStrip0.getRGB();
color1 = tricolor.colStrip1.getRGB();
color2 = tricolor.colStrip2.getRGB();
```

This means that the colors are unique for each page in the site, and these colors tell you something about where you are in the navigation. In fact, if you look at the finished site (you can find the link to this site from the friends of ED site, www.friendsofed.com), you may notice that everything else is tied in to the navigation: the colors, the text—in fact, *everything except the content*! There are no little graphical extras to make it look nice, and this is part of its charm. The pared-down design has a very modern feel to it (the fact that the core UI comes in at about 12 to 15K once it's on the Web is also cool).

As you navigate through the FLA with the new code in place, you'll notice two things:

- You've forgotten to bring the text back once the new page is created.
- If you navigate too far into the site, the colors start to break up, and you lose the graded color scheme.

You'll sort out the first problem in a moment. The second problem is something you're seeing because you're currently allowed to go much further into the navigation than would normally be expected. Usually the **three-click rule** applies: if the user can't get where he or she wants to go within three clicks, your site has poor navigation. You'll arrange the content later so that all pages can be reached within three clicks, and users will never see the color breakup.

The technical reason for the breakup is that you've overflowed on one of the color channels. One of the channels (blue or green, because those are the ones you're changing in the transition) has gone over 0xFF. You lose the subtle tonal changes when this happens because you've suddenly shifted to a different position on the color spectrum.

Finishing the text transition

Bringing the text back again is remarkably easy because you already have the function to fade text (fadeText). All you have to do is set up the variable you used to fade the text out so that this time you fade back in. There are two places you need to add new code. First, you need to add some code to the end of posTransition():

```
    selectedStrip = this._name.substring(5,6);
    selectedCol = _root["color"+ selectedStrip];
    tricolor.setCol(selectedCol, selectedCol, selectedCol);
    fadeColor = 0;
    fadeAlpha = 0;
  }
```

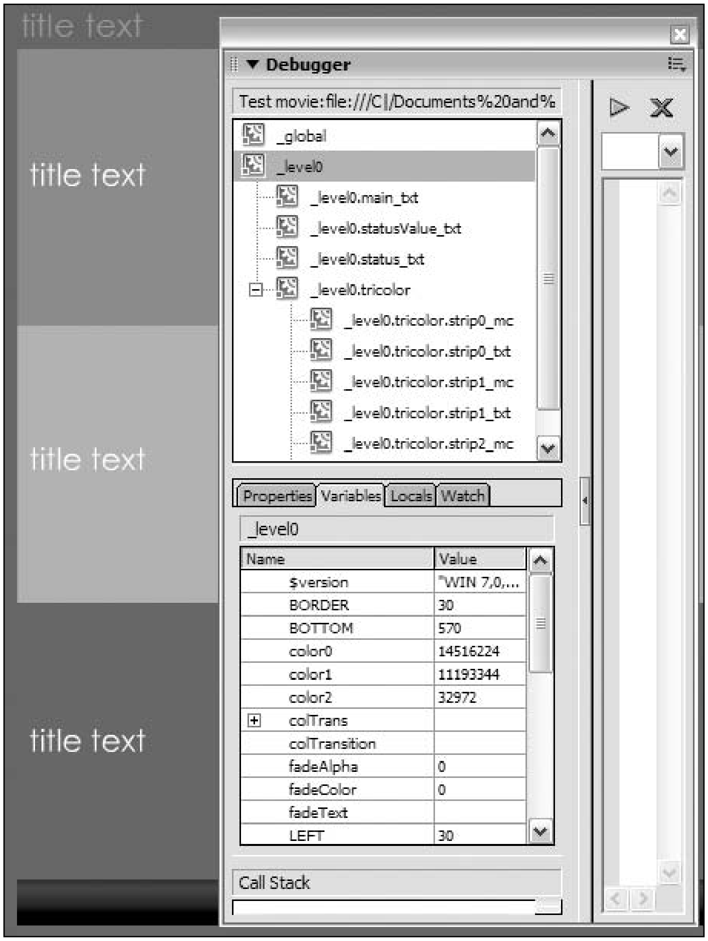
Second, you need to add the code to actually create the transition via the function colTransition():

```
  function colTransition():Void {
    fadeColor += 0x000202;
    fadeAlpha += 6;
    fadeText(fadeAlpha);
    if (fadeColor<colTrans[0]) {
      tricolor.colstrip0.setRGB(selectedCol+fadeColor);
    }
    if (fadeColor<colTrans[1]) {
      tricolor.colStrip1.setRGB(selectedCol+fadeColor);
    }
    if (fadeColor<colTrans[2]) {
     tricolor.colStrip2.setRGB(selectedCol+fadeColor);
    } else {
      // we're now at the new page...
      color0 = tricolor.colStrip0.getRGB();
      color1 = tricolor.colStrip1.getRGB();
      color2 = tricolor.colStrip2.getRGB();
      fadeText(100);
      delete this.onEnterFrame;
    }
  };
```

At the end of colTransition(), you explicitly set the text alpha value to 100. This is to make sure that you end up with fully faded-in text even if you make the other transitions quicker (in which case the alpha fade wouldn't finish in time).

Reviewing the code so far

If you look at your FLA in test mode (or open futuremedia_code03.fla from the chapter download), you're able to see the full color transition of the UI and how the variables and events change as the animations progress. Test the FLA in debug mode and take a look at the Variables tab of root (_level0). You'll see that the various color and fade variables change in two distinct ways as posTransition() and colTransition() are at work. It's almost like watching a clock mechanism!



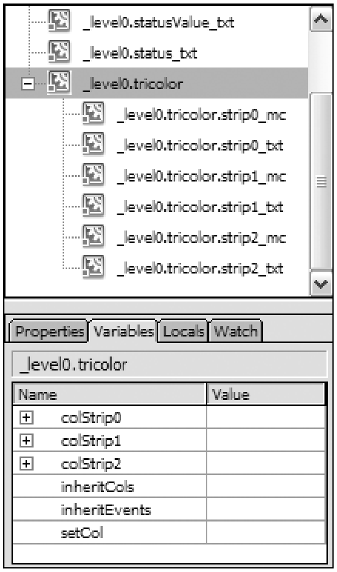
Test movie:file:///C|/Documents%20and%

_global
_level0
_level0.main_txt
_level0.statusValue_txt
_level0.status_txt
_level0.tricolor
_level0.tricolor.strip0_mc
_level0.tricolor.strip0_txt
_level0.tricolor.strip1_mc
_level0.tricolor.strip1_txt
_level0.tricolor.strip2_mc

_level0

Name	Value
$version	"WIN 7,0,...
BORDER	30
BOTTOM	570
color0	14516224
color1	11193344
color2	32972
⊞ colTrans	
colTransition	
fadeAlpha	0
fadeColor	0
fadeText	
LEFT	30

> *When using the debugger, you may find that the variables change in a blur of activity. A good trick is to temporarily change the frame rate to 1 fps to 5 fps when debugging so that you can see the values changing.*



_level0.statusValue_txt
_level0.status_txt
_level0.tricolor
_level0.tricolor.strip0_mc
_level0.tricolor.strip0_txt
_level0.tricolor.strip1_mc
_level0.tricolor.strip1_txt
_level0.tricolor.strip2_mc
_level0.tricolor.strip2_txt

_level0.tricolor

Name	Value
colStrip0	
colStrip1	
colStrip2	
inheritCols	
inheritEvents	
setCol	

You can also see the true relationship between `tricolor` and the three strips inside it by checking the Variables tab for their timelines.

The `tricolor` movie clip is simply a carrier of the three strips and the background functions to drive it all. The individual `strip` movie clips control and initiate the animation (via the `onRelease` events, which are always attached to the strips), and control the `onEnterFrame` events that become attached to the `strip` (you'll see them as `_level0.tricolor.strip0_mc` to `strip2_mc` in the debugger) that was clicked once animation takes place.

Notice also that the `onEnterFrame` events that control the animation disappear as soon as the animation navigation has completed. So what does Flash do when there are no animations going on?

Nothing! It just sits there, waiting for a mouse click on one of the three strips. That's actually a *very good thing*. Some Flash designers blame the Flash development team or sluggish graphics, but that's usually because their UI code is always running in the background, and this makes the final content sluggish. The Futuremedia UI takes up none of Flash Player's precious performance budget when it isn't doing anything, and that makes the site processor-friendly. It will never be the cause of performance bottlenecks that the final content may experience. Flash UIs that are always animating even when users aren't interacting with them are cool as long as they know when to stop and get on with the important task of providing quick and responsive navigation, and then let the content do its stuff once it's reached.

Speaking of content and site structure, the site so far is very colorful. It has the look of a Pantone swatch book (the kind of book graphic designers look at and say, "Wow, that looks kind of cool—if only I could actually do something that looks as colorful as that!"). Guess what, that was one of the influences of this site. However, you don't really seem to be going anywhere in this site. You always end up at pages marked *title text*.

In the final part of this section, you'll turn your attention to adding some structure to the navigation. Unlike the sort of beginner Flash sites you may have tried before now, you'll notice that this site doesn't move up and down a timeline when jumping between pages. Instead, it stays on the same frame (frame 1) and changes the appearance of that frame. To change the pages to include content and titles, you need to define one more major part for the site: **data**.

Data-driven sites (and why you need to understand them)

A beginner **timeline-driven** site is easy to understand because the concepts behind it are simple to grasp. The Futuremedia site is a **data-driven** site. Rather than changing the appearance of the site by moving up and down a timeline, you'll get the UI to look at a set of data that defines how the new page should look.

Although the way this second type of site works is harder to grasp, it's easier to maintain once built. To change pages in a timeline-driven site, you have to physically change the timeline and UI by adding new navigation buttons and keyframes. In short, you have to change everything. To change pages in a data-driven site, you can leave the UI alone—you just change the data associated with the content.

This is your old friend modularity at work again. In a data-driven site, you have two parts (modules), both of which are separate from each other:

- UI
- Data and content

The user requests a page via the interface, and the interface uses this click to do the following:

1. Animate itself to provide feedback.
2. Retrieve the data associated with the requested page.
3. Use the data to build the content associated with the page.

Thus far, you've only looked at creating step 1, which is why the Futuremedia site still looks like it's missing a lot of details that a full site should have.

The UI is thus separate from the data and content (and, incidentally, the data and content are separate from each other because they're different; one is code based, and the other consists of graphics and text), which means that if you want to add new pages, you don't have to change the UI. You instead change just the parts of the module that actually need to change—that is, data and content.

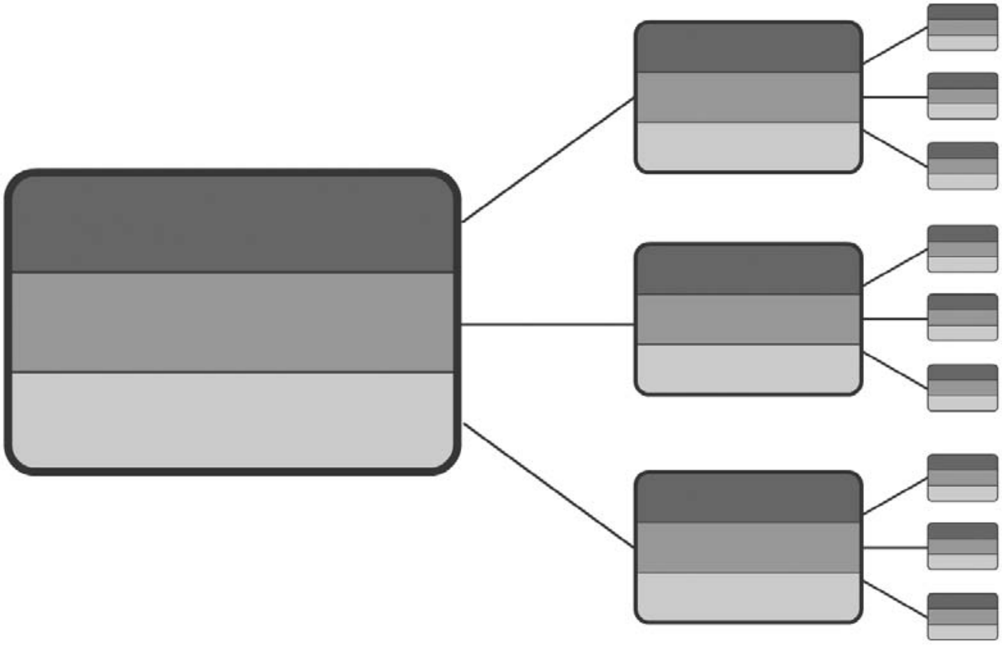
Not only are data-driven sites much easier to maintain than timeline-driven sites, but such sites are also much better in cases where the content can't be known when you build the UI. This "can't be known" bit is usually because the content is **dynamic**, or changeable from day to day. If we conducted a poll of Flash web designers, we would find that the people making the real money are the ones who can build dynamic sites (also called **web application front-ends** or **rich Internet applications**).

The Futuremedia site is built the way it is because we want you to be able to take the concepts taught in this book and actually make money. That's why Futuremedia is a data-driven site. It doesn't get its data from a server (as a real web application would, although you can easily modify it so it does), but it's still an important type of site for your development into a designer who can build Flash sites that further your career. It's the stepping-stone that takes you from timeline-based sites to web applications.

Now that the preamble is out of the way, let's get on with it and get your hands dirty.

Defining data for Futuremedia

The whole site navigation so far looks like this:



There's a home page, and clicking any strip zooms you into a new page, which then magically splits again into another tricolor. This goes on forever, taking you down an endless series of three-way crossroads. You need to add some information here to tell Flash how to do this in a structured way that allows you to do the following:

- Add unique titles to each page.
- Form definite paths. This includes the ability to close off certain paths where you don't want all three strips to take you somewhere.
- Allow you to add a new type of page that doesn't turn into more `linkPages`, but enables you to display content (`contentPages`).

And while you're at it, you might as well push the boat out and add some other cool stuff that normal tween-based sites can't do. You'll make the content general with the following:

- The ability to define the content as something that's currently in the Library
- The ability to define content that needs loading as a SWF file

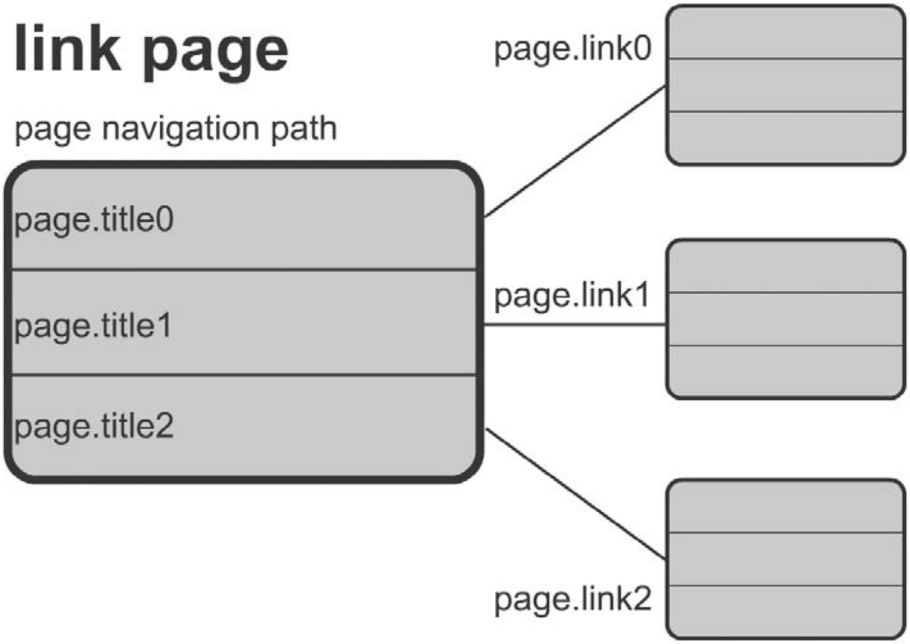
Wow! That's a lot of stuff. There are two ways you could do this. One way is by defining a set of arrays for each full page. The other way is by having all the information for the full site in one place. You're moving forward in leaps and bounds, so you'll go for the latter, because you're feeling confident about all this.

> *There's a third way you could do this: you could define the data as an XML file. We assume you won't know much about XML at this stage, so you won't go down this route, but there's nothing stopping you from doing so later.*

To define the required information of each page, you need to set up a **record** per page. Within each record, you'll have **fields** that define each page, covering all the preceding information in a structured way.

> *The Futuremedia site is structured on at least two levels. On one level, the code itself is written in a structured way using functions, and on another level, the data that drives the content is itself also structured.*

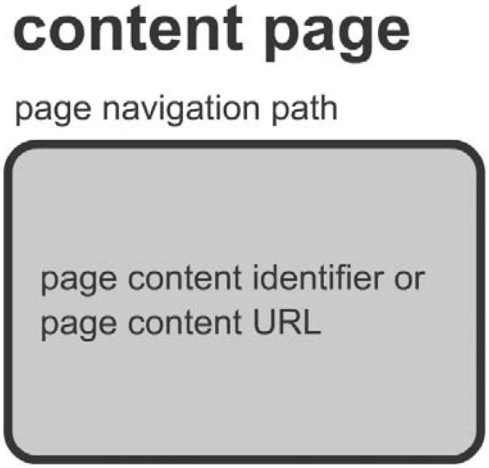
So exactly what bits of information do you need to know? Well, for the pages you've seen so far (pages that link to other pages, or `linkPages`) you have this:



You need to know the following:

- Three items that define links to other pages.
- Three page title strings plus one page navigation path. You can either get the UI to work this last one out or just put up a string that you've defined (you'll go for the latter).

If the page will show some content, you'll want another game plan that looks something like this:



Here, you're more concerned about the content you'll show. You need to know the following:

- The movie clip linkage identifier if you're going to pull content from the Flash Library
- The URL that the content lives at if it's a SWF

Now here's the big question. If you want to represent all this information as a *general* set of data that Flash could understand, how would you do it? Well, you know about arrays from Chapter 3, but you couldn't really use them, because arrays store only *lists*. You need something more structured. The answer is that you need to produce an *object*.

As you learned earlier in the book, objects have properties that describe them. Properties are just raw data, and it's up to you to decide what the properties mean. You have some information you want to define for the site, so how about you do that? **Property** is just a fancy programmer way of saying "something that describes a thing." You have a bunch of things that you want to describe on your web pages, so how about you do that with this odd **object** thing?

You need several properties, but there's a problem. You have two kinds of pages! How do you do it? Well, you can have two objects, one for the content pages and one for the link pages. That isn't how programmers think, though—programmers like to *generalize*. In this context, "generalize" means "what you have to do for there to be only one kind of page." See if you can guess without peeking at the answer . . .

The answer is *you make the page type a property as well*!

So you'll have something called page with the following properties (all of which we've taken from the previous diagrams):

- Page type
- Page links
- Page titles
- Page content linkage identifier
- Page content URL

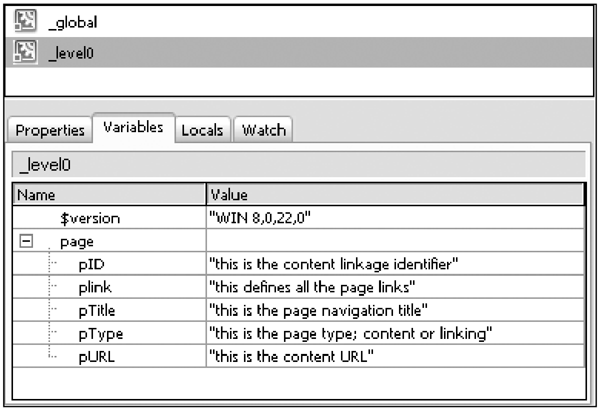
And that's everything you need to define your site's pages. In fact, in principle, it's almost enough to define *any* web page!

> *The final product of this little trial is included in the download as* `futuremedia_database.fla`.

Let's create an object in Flash that has these properties. Open a new FLA (you can do this even if you have something else open in Flash), select frame 1 of the timeline, and type the following code in the Script pane of the Actions panel:

```
var page:Object = new Object();
page.pType = "this is the page type; content or linking";
page.plink = "this defines all the page links";
page.pTitle = "this is the page navigation title";
page.pID = "this is the content linkage identifier";
page.pURL = "this is the content URL";
```

Test the FLA in debug mode (Control ➤ Debug Movie). Click the Continue button (represented by the green "play" icon) to start the SWF, and then select `_level0` in the top-left pane. Click the Variables tab to see the data structure you've just created.



Name	Value
$version	"WIN 8,0,22,0"
⊟ page	
pID	"this is the content linkage identifier"
plink	"this defines all the page links"
pTitle	"this is the page navigation title"
pType	"this is the page type; content or linking"
pURL	"this is the content URL"

Now you're probably thinking, "That's fine, but some of the properties of my pages have properties of their own. For example, the `pLink` part will need three links, one for each strip."

So you're stumped, right? Well, sort of. There are two ways to fix this problem. One way is to build a constructor function that creates subproperties. The other, much simpler, way is to use the fact that properties can be *any* data type themselves, and that includes strings, numbers, Booleans, and arrays.

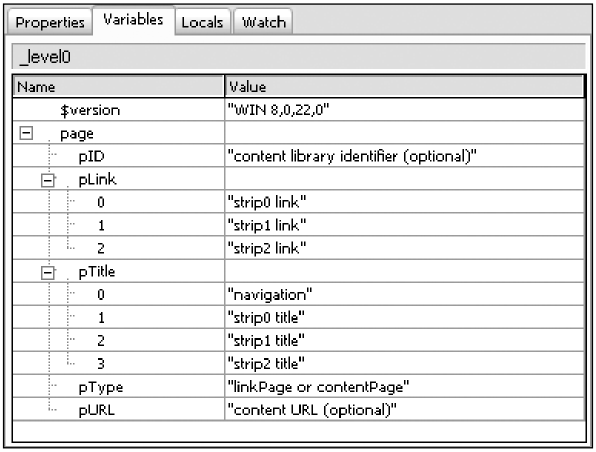
If you go through and define your properties as the actual *types* you need, and make those that contain multiple subproperties into arrays, you can encapsulate all your descriptive data for a single page into a single object. Let's do that now. Change the code as follows:

```
var page:Object = new Object();
page.pType = new String();
page.plink = new Array();
page.pTitle = new Array();
page.pID = new String();
page.pURL = new String();
```

This gives a structure to your properties, but it doesn't put in any data. You should do that next. Add the following lines to what you have so far:

```
// add the data for a general page...
page.pType = "linkPage or contentPage";
page.pLink = ["strip0 link", "strip1 link", "strip2 link"];
page.pTitle = ["navigation", "strip0 title", "strip1 title", ➥
 "strip2 title"];
page.pID = "content library identifier (optional)";
page.pURL = "content URL (optional)";
```

Now run the FLA in debug mode again. This time you'll see a much larger structure of properties. In fact, there are enough branches on the tree to fully define the two kinds of pages. You have all the titles, the locations from which to get the content, and the list of links (including the back button link) of each page.



_level0

Name	Value
$version	"WIN 8,0,22,0"
page	
pID	"content library identifier (optional)"
pLink	
0	"strip0 link"
1	"strip1 link"
2	"strip2 link"
pTitle	
0	"navigation"
1	"strip0 title"
2	"strip1 title"
3	"strip2 title"
pType	"linkPage or contentPage"
pURL	"content URL (optional)"

In the same way that the folder in which your doctor keeps your medical history is called your medical record, this structure is a page record. Your medical record defines your health history since birth. The Futuremedia site's page record fully defines the page.

You have one last problem. You have only one record, but you have many pages in your site. You need some way of building more of these records. The doctor's staff can look up a patient's medical history in a filing cabinet of medical records before the patient is treated, and you need to be able to do the same for the current page the UI needs to display, so that Flash knows what to show, where to link to, when a strip is clicked, and so on for all the properties of every page.

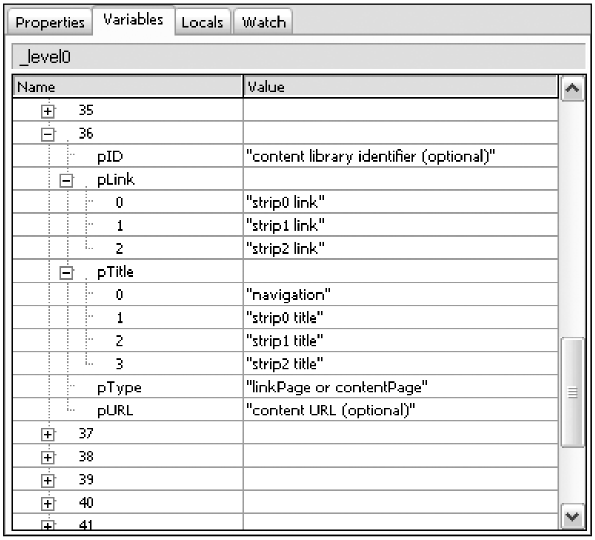
Suppose there are 50 pages. You need 50 of the previous structures. How many lines do you think that you'll have to add to your listing so far? Guess!

OK, it's not many. Handling large quantities of repetitive data (or, to use a polite programming term, **structured data**) is something computers are very good at. You have to add only three new lines (and one of them is a brace, so it's really two lines!). You also have to change all references to page to page[i]:

```
var page:Array = new Array();
for (var i:Number = 0; i < 50; i++) {
  page[i] = new Object();
  page[i].pType = new String();
  page[i].plink = new Array();
  page[i].pTitle = new Array();
  page[i].pID = new String();
  page[i].pURL = new String();
  // add the data for a general page...
  page[i].pType = "linkPage or contentPage";
  page[i].pLink = ["strip0 link", "strip1 link", "strip2 link"];
  page[i].pTitle = ["navigation", "strip0 title", "strip1 title", ➥
 "strip2 title"];
  page[i].pID = "content library identifier (optional)";
  page[i].pURL = "content URL (optional)";
}
```

So what have you done? Well, you created an array of size 50 called page. You made each array element, page[i], an object, which allows you to add the properties of page number i.

Cool! So now you have a data structure (it's actually a simple **database of records**) that you can use to make Flash create all your page titles and links. You can see it in the debugger if you debug the FLA one last time. You'll see that there are now 50 sets of records (0 to 49). If you open any of them, you'll find the familiar page-definition structure.



```
_level0
Name                 Value
  + 35
  - 36
      pID            "content library identifier (optional)"
    - pLink
        0            "strip0 link"
        1            "strip1 link"
        2            "strip2 link"
    - pTitle
        0            "navigation"
        1            "strip0 title"
        2            "strip1 title"
        3            "strip2 title"
      pType          "linkPage or contentPage"
      pURL           "content URL (optional)"
  + 37
  + 38
  + 39
  + 40
  + 41
```

Going back to the medical record analogy, you've just created a filing cabinet full of records. Your patients are labeled 0 to 49, and you've filed the records numerically. At the moment, each record is filled with basic default information, but you'll soon make the records specific to each page in your site.

You may recognize the preceding tree structure as something you could define via a separate XML file that the final SWF will load at runtime. You've come a *very* long way here!

In the book project section in Chapter 12, you'll integrate the page data structure into the Futuremedia site, and then you'll start to use it to produce real navigation.

Parting shots

We've thought long and hard about building a site that was based around a set of records. Many existing Flash designers would see this as advanced.

Times have changed since the early days of Flash, when a nice bit of Flash eye candy would have been termed "cool." In the modern Flash world, you have to look beyond graphics to build a cool website. You need to have structured and extensible code, and sites that are easy to manage.

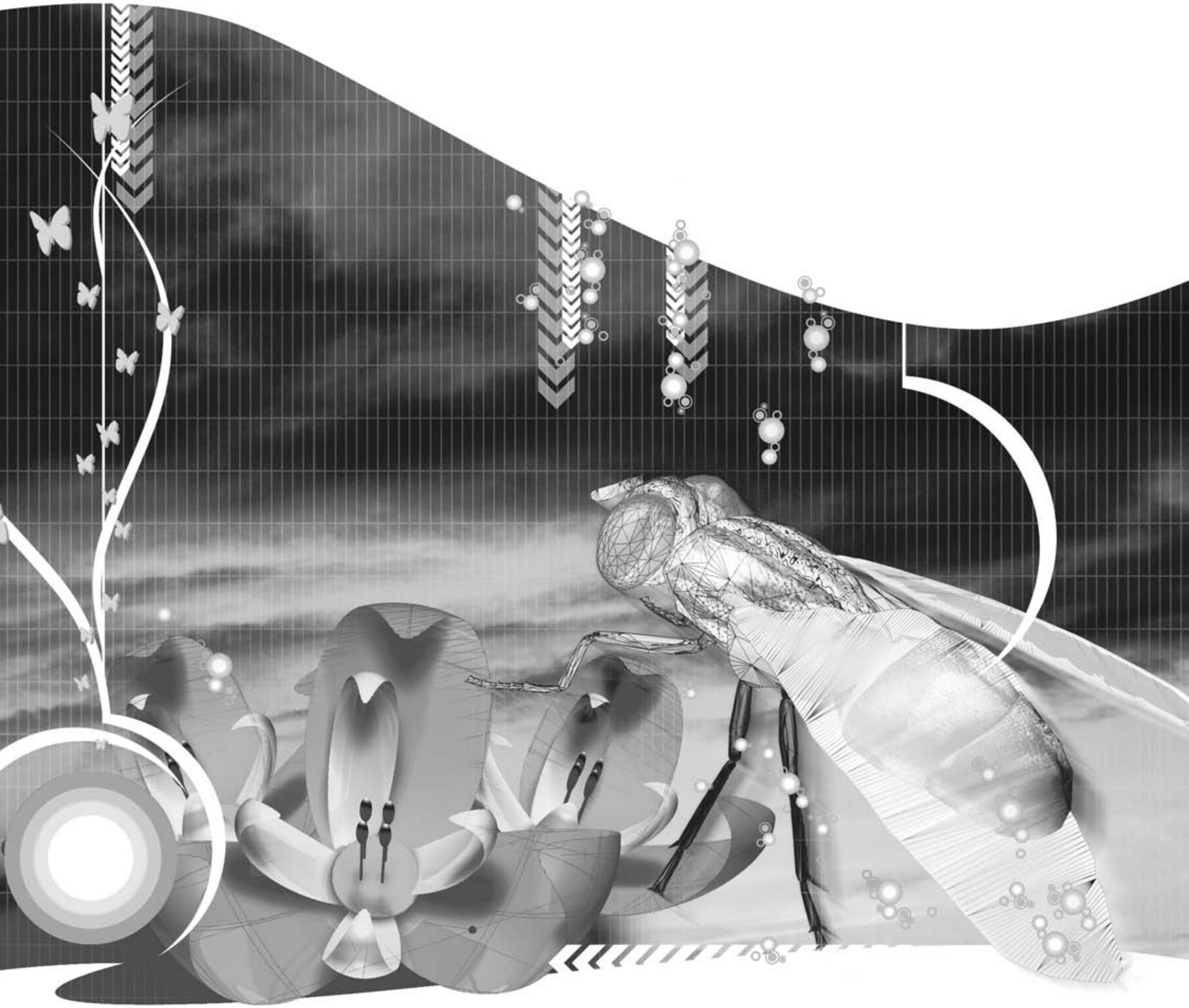
Something you may have noticed is that you've used almost all of the information you've acquired so far in the book: variables, objects, loops, arrays, and even that inertial effect that you thought would be of use only in a "for fun only" flies-chasing-the-mouse-pointer FLA (which has actually become a fundamental part of the Futuremedia site—the site zooms with the same inertial effect!). Like the approach to the final scene of a movie, all the elements of the plot are finally starting to come together.

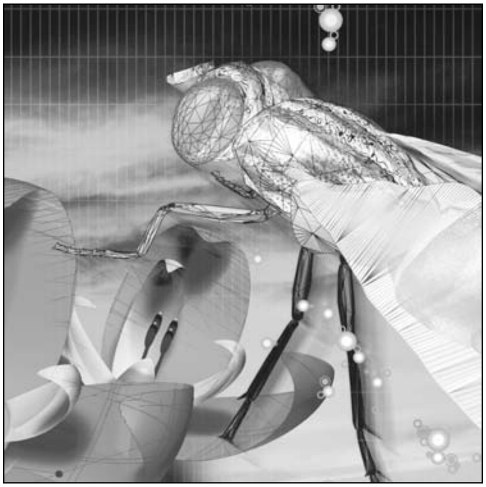
Summary

You've learned in this chapter that by using the drawing API, it's possible to *create* motion graphics via ActionScript, not just *manipulate* graphics that you previously created in Flash.

It's somewhat ironic that you started out with Flash working in a purely visual way, with all tweens with no code. And yet here you've produced quite impressive visual effects purely by using ActionScript—there's nothing else in the FLAs. When you began this book, you probably never thought you'd end up here!

In the next chapter, you'll finish up the data structure that you've defined in this chapter. You'll populate the structure with real data, and you'll start using it to drive the site, finally making the Futuremedia site truly "data driven."





Chapter 12

ADDING SOUND TO FLASH

What we'll cover in this chapter:

- Choosing the right sound format
- Importing sound clips into the Library
- Converting a sound clip into a sound object that ActionScript can control
- Using event handling to synchronize sounds
- Adjusting sound objects to pan and change the volume of a sound effect
- Streaming audio
- Giving visitors the option to turn off sound

Clever use of music and sounds in your designs allows for a high degree of interactivity and usability, and at the very least produces a pleasant atmosphere within your sites. With Flash, you can now begin to push the limits of using sound in your websites, from streaming sound in parallel with video to actually sequencing and mixing completely new pieces of music online.

> *Flash Player 8 allows you to use 32 simultaneous channels of audio in your movies—four times as many as previous versions. So if audio is what turns you on, you'll be able to use a whole lot more of it now, as long as your target audience is using a recent version of Flash Player.*

In the past, sound on the Web was problematic, but then **MP3** (MPEG Audio Layer-3) came along. MP3 compresses sounds intelligently to give a much smaller file size for a given quality than any other sound compression system currently available. It's now *the* standard for music files on the Web, in the same way that Flash is the standard for high-impact, highly interactive websites. Before the advent of MP3, there were limited options for the web designer: **Musical Instrument Digital Interface** (MIDI), which isn't a universal Web standard, and **sampled sound files** of various formats (like WAV and AIFF), which can be rather bulky—a major headache during download!

By default, Flash saves sounds in MP3 format, greatly reducing the size of your movies. However, it's important to understand some basic principles about the different formats before adding sound to your Flash movies.

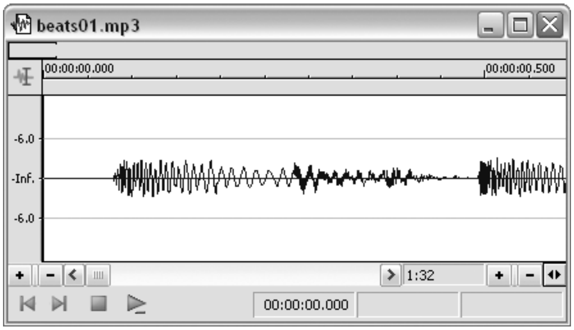
Choosing the right sound format

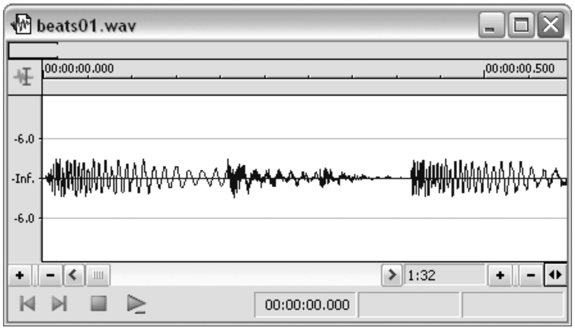
Of all the sound formats currently available, MP3 is the only one that you can import directly into any version of Flash 8. However, if you have QuickTime 4 or later on your computer, you can import several other sound formats into Flash, including the two most popular formats: **WAV** (waveform sound format) and **AIFF** (Audio Interchange File Format). Still, if MP3 is now *the* standard for the Web, why would you want to use anything other than MP3? Importing sound from MP3 files has two disadvantages:

- MP3 is what's known as a **lossy** format. When storing the sound, the computer discards sounds that it thinks you won't hear; and once the information is discarded, it's gone forever.
- MP3 files leave a tiny amount of silence at the beginning and end of the file, so you can't use them for seamless sound transitions or looping.

The first issue is something that you should be familiar with from storing images. The JPEG format is also lossy. Although changing the quality setting of a JPEG file can reduce its size considerably, the image quality gets progressively worse, and there's no way of restoring what may have been a sharp image from a low quality JPEG. It's exactly the same problem with sound. It's far better to preserve your original sounds in the highest possible quality. Usually, this means keeping sounds as WAV or AIFF files, and importing them directly into Flash in that format. Flash will then perform the conversion to MP3 when compiling the SWF.

The following two screenshots show the first half-second of the same sound as viewed in an audio editing program. The file on the left has been saved in MP3 format and the one on the right as a WAV file.





As you can see, the sound wave in the MP3 file starts off completely flat. This represents approximately one-twentieth of a second of silence. A similar period of silence is also added at the end of the sound. As a result, you get a tiny but noticeable gap between sounds if you attempt to use MP3 files in a loop or a consecutive sequence of sounds. On the other hand, the sound wave in the WAV file begins right at the start and finishes right at the end. This results in nice, smooth transitions (assuming you have edited the sounds properly) and no gap.

Of course, if you have a high-quality MP3 file, and it doesn't need to loop, there's nothing wrong with using it. The point to remember is that Flash normally uses MP3 for replay, but the original file can be in any format compatible with Flash.

> *For a full list of compatible sound formats, see* Help ➤ Flash Help ➤ Using Flash ➤ Working with Sound ➤ Importing sounds.

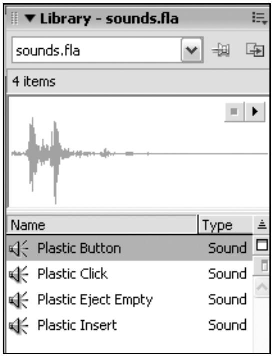
Using sound on a timeline

Working with sound in Flash can be quite complicated. In fact, there are so many options to choose from that even advanced Flash users often don't know when to use one over another. Don't worry, though—you're going to start from the absolute basics and build up gently from there. We won't go into every detail, but by the end of the chapter you should have a good enough grasp of the basics to start exploring the more advanced options yourself.

Attaching a sound directly to a timeline

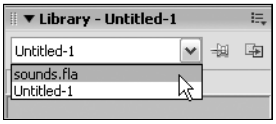
The simplest way to add sounds to a Flash movie doesn't involve using ActionScript at all. It's fairly limited in terms of what it lets you do, but let's use it as a starting point so that you can see what's going on.

1. Open the download file sounds.fla and open the Library panel, if it isn't already. Sounds, like all other assets for a movie, are stored in the Library. If you select the first item in the Library panel, you will see a **sound wave** in the preview panel, showing you roughly how the sound's volume varies over time. To hear what it sounds like, click the play button in the top-right corner of the preview panel.

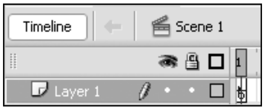


Name	Type
Plastic Button	Sound
Plastic Click	Sound
Plastic Eject Empty	Sound
Plastic Insert	Sound

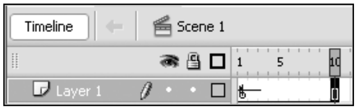
2. You can attach the sounds to any keyframe you like. Open a new Flash document, and then display contents of the sounds.fla Library by selecting sounds.fla from the drop-down menu at the top of the Library panel.



3. Select the item called Plastic Button and drag it onto the stage. You won't see anything appear on the stage itself, but take a careful look at frame 1 on the timeline, and you'll see a small horizontal line running through the middle.



4. Insert a frame (F5) at frame 10, and you'll see that this horizontal line runs right across the first few frames of the timeline—it's actually a tiny picture of the sound wave you just attached to frame 1.



5. The sound will play whenever the playhead hits the keyframe it's attached to. As you can see from the timeline, the Plastic Button sound will begin playing when frame 1 is entered, and it will finish playing by frame 3. Test the movie now, and you should hear the sound repeated roughly once a second as the movie loops around its ten frames.

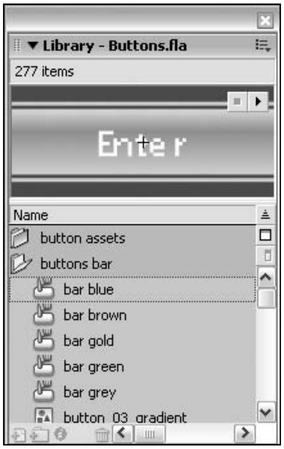
Attaching sounds to button keyframes

You can easily attach sounds to keyframes on a button. If you attach a sound to the *down* state, it will play every time the button is clicked.

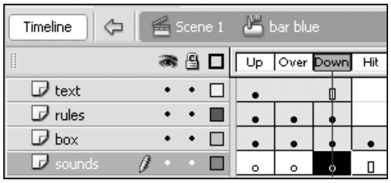
1. Keep sound.fla open, but close the other file you've just been working with. There's no need to keep the previous one unless you want to drive everyone mad with the relentless clicking. Open a new Flash document.
2. Use the drop-down menu in the Library panel to switch back to the sound.fla Library. Highlight Plastic Button and Plastic Click, and press Ctrl+C/Cmd+C to copy them.

3. Switch back to the new document's Library, click inside the Library panel, and press Ctrl+V/Cmd+V to paste the sound files. You should now have copies of both sound files in your new document.

4. Select Window ➤ Common Libraries ➤ Buttons and drag an instance of any button that strikes your fancy onto the stage—bar blue from the buttons bar folder will do just fine.

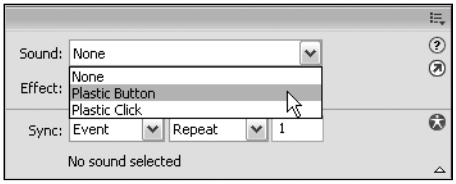


5. Double-click the button to edit it in place. Add a new layer called sounds at the bottom of the timeline, and insert keyframes in the Over and Down frames on the new layer.



6. At this point, you could easily drag sounds from the Library straight onto the stage—like you did before—to attach them to the currently selected keyframes. However, there's another way to do it that's more precise and allows you to *remove* sounds from keyframes as well as add them.

   With the Over keyframe selected, take a look at the Property inspector. You'll see a drop-down menu labeled Sound. Select Plastic Button from the Sound drop-down menu to attach it to the keyframe.


Sound: None
Effect:
Sync: Event Repeat 1
No sound selected

7. Repeat step 6 for the Down keyframe, selecting the Plastic Click sound.

8. Test the movie. You'll hear the two sounds play as you roll over or click the button. It's not normal practice to add sounds to the Up state, and attaching sounds to the Hit state is effectively the same as attaching sounds to the Down state (so you need to attach a sound to only one or the other, not both).

None of this should really be new to you, but it does serve to emphasize an important concept: sounds must always be *attached* to something. Attaching sounds to keyframes on a timeline is a simple way to put them in your movies, but it's fairly inflexible.

ActionScript gives you a powerful and more versatile approach. You can attach sounds dynamically at runtime, and you can attach sounds to non-timeline-based events (such as pressing a key or a mouse button) or any more complex events that your code has been wired up to recognize (such as "site has finished loading" or "spaceInvader12 has just been hit"). Let's start making some noise with ActionScript.

Using the ActionScript Sound class

This is where you step into new territory and start using **sound instances** so that you can truly program your sound effects. Sound instances are the fundamental building blocks that you use to work with sound in ActionScript. There are three simple steps involved in using a sound instance in your movie:

1. Define a sound object.
2. Attach sound data to the sound object.
3. Use the sound object's methods to control how and when the sound is produced.

As you'll learn shortly, these methods give you some pretty neat ways to control sound. For instance, if you have a car moving from left to right on the screen, you can write some ActionScript to make the car's engine sound pan from left to right at the same rate. In the same way, if the car moves into the distance, ActionScript can make the sound fade.

All of this is possible in Flash, and you even have some other sound features to play with:

- The `onSoundComplete` event for detecting when a sound has finished playing. This allows you to do something called **sound sequencing**, which allows you to create (among other things) interactive mixing boards.
- The `loadSound()` method for loading sound files directly into movies at runtime to create, for example, a Flash jukebox.
- The ability to **stream** audio files. Streaming allows users to hear the sound while it's being downloaded. Since audio files tend to be very large, this usually avoids annoying delays (although streaming doesn't always work well on a slow connection).

In the next section, you'll learn how to create a sound instance and use it to vary sounds dynamically.

Playing sounds from ActionScript

With ActionScript, you can make all sorts of subtle changes to your sounds, perhaps in response to user input, *while the movie is running*. That doesn't mean you don't have to make any design decisions—quite the contrary, in fact. It means that you can design complex, flexible soundtracks that can adjust themselves to all sorts of different situations.

The possibilities are truly fantastic. In fact, the only way to convey the scope of what this means for your movies is to show some of those possibilities being put into practice. So, once you've walked through the basics, that's precisely what you'll do.

Attaching sounds to a sound instance

In the last exercise, you attached sounds to keyframes on the timeline. This time, you'll attach them to a sound instance, so your first step is to create that sound object.

Creating a sound instance and attaching a sound

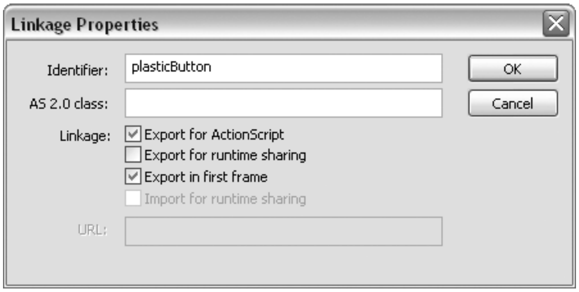
1. Open the file `sounds.fla` yet again, and save it as `soundObject1.fla` (alternatively, take a look at `soundObject1.fla`, where you'll find the completed code for this example). Rename Layer 1 as actions and select frame 1. Pin the Actions panel to this frame and put the following ActionScript on the first line of the Script pane:

```
var mouseSound:Sound = new Sound();
```

2. Next, you must attach a sound file to this object. There are two ways to do this: from the Library with the `attachSound()` method, or directly from an external file. Let's concentrate on the first approach for now, as it's similar to working with movie clips. We'll take a look at loading from external files later in the chapter.

   Sounds are exactly the same as movie clips in that they need to be given a **linkage identifier** before you can call them up from your code. Select Plastic Button in the Library panel, right-click/Ctrl-click, and select Linkage from the context menu. In the Linkage Properties dialog box, select the Export for ActionScript check box. This will automatically also select the Export in first frame check box (don't uncheck this box!) and make the Identifier and AS 2.0 Class text input boxes active. You're interested only in the Identifier field; leave the AS 2.0 Class field blank.

   Flash automatically enters Plastic Button in the Identifier field. Because a linkage identifier cannot contain any spaces, change the Identifier text to plasticButton and click OK.


Linkage Properties

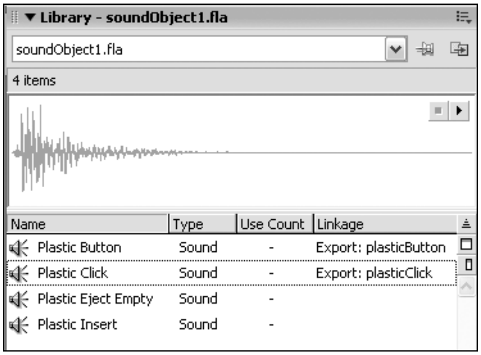
Identifier: plasticButton

AS 2.0 class:

Linkage: ☑ Export for ActionScript
□ Export for runtime sharing
☑ Export in first frame
□ Import for runtime sharing

URL:

3. Create a linkage identifier for Plastic Click and use plasticClick as the identifier name. In the Library panel, you should now see the identifiers appear in the Linkage column, as shown in the screenshot. You may need to increase the width of the Library panel or scroll horizontally to see this column.


▼ Library - soundObject1.fla

soundObject1.fla

4 items

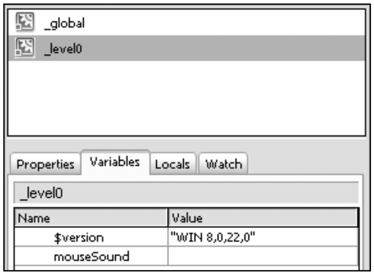
Name	Type	Use Count	Linkage
Plastic Button	Sound	-	Export: plasticButton
Plastic Click	Sound	-	Export: plasticClick
Plastic Eject Empty	Sound	-	
Plastic Insert	Sound	-	

For every sound clip you use, you have three names to consider: the name of the file the sound came from, the name the sound is given in the Library, and the linkage identifier. It's this last name that's important as far as ActionScript is concerned.

The reason you have to check the Export for ActionScript *and* Export in first frame *check boxes is because the Flash compiler decides whether you use a symbol or asset by looking for it on the timeline at compile time. However, symbols and assets that are used by ActionScript aren't there; they're placed on the stage dynamically at runtime. So, the* Export for ActionScript *check box is your way of saying, "I actually use this symbol or asset at runtime via ActionScript, so make sure it's exported in the final SWF."*

Flash also needs to know on which frame you need the symbol or asset. For ActionScript, it isn't usually obvious when the code will place the content on the timeline. For example, the frame in which the "all aliens are dead, you have won" sound is required in a Flash video game will depend on how good the user is at playing space invaders. To make sure that ActionScript will be able to load content from the Library as and when required, select the Export in first frame *check box to ensure the content is available at the first frame it can be requested by ActionScript: frame 1.*

4. If you test the SWF, you'll find . . . nothing happens! That's because you've only *created* a sound instance. You can't hear the instance because it has nothing to start playing, and you can't see it either because . . . uh, it's a sound. You *can* see its data representation, however, if you use the debugger. To check for this, select Control ➤ Debug Movie and then start the debug session by clicking the play icon at the top right of the Debugger panel. You'll see the sound instance you've just created at the correct scope (look for it on _level0 under the Variables tab).



Name	Value
$version	"WIN 8,0,22,0"
mouseSound	

5. Back to the code. You need to attach the Flash Library's sound clip to the ActionScript sound object. Close the movie preview and Debugger panel, if you haven't already done so, and add the following as the second line of your code:

```
mouseSound.attachSound("plasticButton");
```

The argument for attachSound is a literal string (you know this because it's enclosed in double quotes), and this means it must be **exactly** the same as the linkage identifier you just defined in the Linkage Properties dialog box. (Remember, ActionScript 2.0 is case sensitive.)

One of the most off-putting things about ActionScript-based sound is that if you get this step wrong, the sound file won't be attached to the sound instance, and you won't hear anything when you try to start the sound. Worse still, Flash doesn't tell you if this has happened; it is what's called a **silent failure**.

6. The only way to tell if a sound has been correctly set up is to start it. Add the new line 3 as follows:

```
var mouseSound:Sound = new Sound();
mouseSound.attachSound("plasticButton");
mouseSound.start();
```

   This new line causes the sound to start. It will play once, and you should hear a single click when you test the movie. If you don't hear the click, check that the linkage identifier in the code matches the one in the Library panel.

> *To play a sound, you use* mySound.start(), *not* mySound.play() *(which doesn't work!). Getting into the habit of saying "start the ActionScript sound" rather than "play the ActionScript sound" will help you avoid using the wrong method. Remember:* play() *is for movie clips,* start() *is for sound.*

7. You should now have a working sound instance. At the moment, your sound starts on frame 1, which doesn't really have any advantage over simple timeline sounds. The advantage of ActionScript-driven sound is that it can be interactive and dynamic, so let's add some of those magic ingredients now. Change your code as follows:

```
var mouseSound:Sound = new Sound();
mouseSound.attachSound("plasticButton");
this.onMouseDown = function() {
  mouseSound.start();
};
```

8. Test the movie. You should hear the plasticButton sound whenever you click your mouse button.

Working with longer sounds

Short sounds don't require much control. Longer sounds, though, bring with them problems of their own. To investigate this, import the sound file songLoop.wav (File ➤ Import ➤ Import to Library).

1. Right-click/Ctrl-click the new sound in the Library panel and select Linkage from the context menu. In the Linkage Properties dialog box, select Export for ActionScript, and set the Identifier field to songLoop.
2. Change the code to use the new sound (or use soundObject2.fla):

```
var mouseSound:Sound = new Sound();
mouseSound.attachSound("songLoop");
this.onMouseDown = function() {
  mouseSound.start();
};
```

3. Test the movie. If you press the mouse button now, you'll hear a funky beat. Foot-tapping stuff to be sure, but listen to what happens if you press the mouse button a few times without waiting for the current sound to stop.

   If you input multiple quick mouse presses, you'll hear a number of different versions of the sound. This can produce some cool and unexpected effects, but it isn't really ideal. You need a way to stop the sound just before you start the next one.

Flash Player 8 has ***32-channel sound****, which means you can hear up to 32 mono or 16 stereo sounds at the same time. If all available channels are already playing something, Flash will ignore any new sounds. If you're not careful about stopping unwanted sounds (especially if they're long), you may run out of sound channels and prevent other sounds in your movie from starting at all. In addition, you may get that unwanted echo-noise effect you just experienced, which is something that can make your site appear a little buggy and amateurish when it comes to audio.*

Starting and stopping sounds

Although you can start a sound simply by applying the `start()` method to the sound instance it's attached to, `start()` can also take two optional arguments, like this:

```
mySound.start(offset, loops);
```

The *`offset`* value (in seconds) defines how far into the audio you want to start, and the *`loops`* value defines how many times you want to play the sound. If you specify more than one loop, each one will start at the offset value.

You can stop a sound with the ActionScript command `stop()`. So far, so good, but the `stop()` method of the Sound class works in the following mysterious ways:

- If you use `stop()` without any arguments, **all** sound is stopped in the SWF.
- To stop a particular sound, you must pass its linkage identifier to `stop()`.
- It doesn't matter which sound instance you use with `stop()`.

Starting and stopping sounds cleanly

1. Continue working with the file from the previous exercise. Add the following new lines in the `onMouseDown` event handler (or use `soundObject3.fla`):

```
var mouseSound:Sound = new Sound();
mouseSound.attachSound("songLoop");
this.onMouseDown = function() {
  mouseSound.stop("songLoop");
  mouseSound.start(0, 2);
};
```

The first new line ensures that if songLoop is currently playing, it will be stopped, ensuring that you don't fill all 32 sound channels with duplicates. If songLoop isn't playing, this line will have no effect, so there's no need to wrap it in a conditional clause. The second new line of code will then start songLoop from the beginning (0) and loop it twice (2).

2. Test the FLA again, and you'll find that you no longer get any odd echo effect. If you click the mouse button while songLoop is still running, the stop() command stops it, and the following line starts the two loops again from the beginning.

3. Let's see what happens if you have more than one sound instance in your movie. Change the code like this (it's in soundObject4.fla):

```
var mouseSound:Sound = new Sound();
mouseSound.attachSound("songLoop");
var unusedSound:Sound = new Sound();
mouseSound.start(0, 2);
this.onMouseDown = function() {
  unusedSound.stop("songLoop");
};
```

This adds a new Sound object called unusedSound and moves the start() command back to the main timeline. Finally, the stop() command has been applied to unusedSound.

4. Test the movie, and songLoop should begin immediately. Press the mouse button, and it will stop—even though songLoop is attached to mouseSound and not to unusedSound. The Sound class works in mysterious ways, indeed.

5. What if you want to make sure that the triggered sound plays right through before it can be triggered again (instead of restarting every time the event occurs, whether or not the sound has finished)? You might do this, for example, when you want the user to hear the full sound (if it's a voice giving some information, for instance). It also feels more professional, because you don't get that awful "scratching" interruption effect. Fortunately, Flash gives you the onSoundComplete event, which occurs whenever a sound completes all the loops specified in start().

So, to make your mouse-driven sound play right through following a mouse click (and ignore subsequent clicks until it's done) you need to modify the script to look like this (you can find the code in soundObject5.fla):

```
// set up the sound object
var mouseSound:Sound = new Sound();
mouseSound.attachSound("songLoop");
//
// set up the sound finished flag
var soundFinished:Boolean = true;
//
// set up an onSoundComplete event
mouseSound.onSoundComplete = function() {
  soundFinished = true;
};
//
// set up a handler function for the onMouseDown event
this.onMouseDown = function() {
  if (soundFinished) {
    mouseSound.start(0, 1);
    soundFinished = false;
  }
};
```

As you can see, the code has really started to expand. We've added in comments so that you don't forget what each bit does.

You use a Boolean called soundFinished to indicate whether the mouseSound sound is currently stopped (true if it's stopped; false if it's playing). As soon as soundFinished becomes false, the onMouseDown event can no longer trigger new sounds because the main event handler code is inside an if statement that will run only if soundFinished is true.

This situation will continue until the sound runs through the specified number of loops, whereupon the onSoundComplete event handler sets it back to true, allowing the onMouseDown event to start new sounds.

*This is a very useful bit of code because it contains a commonly used advanced coding structure (known as an **interlock**, **semaphore**, or **inhibit flag**, depending on the context in which it's being used). Essentially, you're using one event (*onSoundComplete*) to modify a value (*soundFinished*) that governs what actions are triggered by another event (*onMouseDown*).*

In fact, this is the same sort of structure that your operating system uses in multitasking. For example, when two programs want to access the same hard drive, the first will lock the second out until it has finished. The fact that each program limits its usage to very short periods of time gives humans the impression that both are actually using the same hard drive simultaneously. Sneaky, huh? As you can probably imagine, it's possible to set up some very sophisticated responses to user events when you organize things in this way.

Creating dynamic soundtracks

While you're looking at the onSoundComplete event, let's do something fun with it. You now have a way to spot when a sound finishes playing, so why not use that event to trigger another sound? You should be able to link several sounds together quite seamlessly and create an ongoing soundtrack that can jump from one sound clip to another (in response to the user or whatever's going on in the movie) without any audible joins or, in other words, a **dynamic soundtrack**.

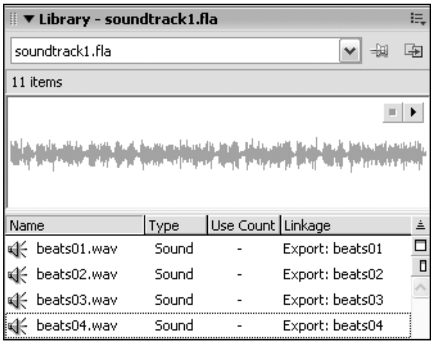
Creating dynamic soundtracks from scratch can be a tricky process, mainly because you need to find (or create) a number of sound clips that will sound OK together, no matter what order you play them in. Fortunately, we've done the hard bit for you with four tracks that dovetail with each other. Because we want them to play seamlessly, they've been saved as WAV files. Admittedly, they're never going to win any Grammy awards, but it's the ActionScript that really matters here.

The actual ActionScript isn't too hard: you basically need to trigger each new sound with the onSoundComplete event that's generated by the last sound finishing. Let's start putting together the movie.

Building the soundtrack player

1. Create a new document in Flash. Change the size of the stage to 250 × 150 pixels. Then use File ➤ Import ➤ Import to Library to import `beats01.wav`, `beats02.wav`, `beats03.wav`, and `beats04.wav` from the download files for this chapter. (If you just want to look at the finished code, open `soundtrack1.fla`.)

2. In the Library panel, right-click/Ctrl-click each sound in turn to set its linkage properties. (Refer back to step 2 of the exercise titled "Creating a sound instance and attaching a sound" if you need a reminder of how to do it.) Name the identifiers beats01, beats02, beats03, and beats04. When you have finished, the Library panel should look like the screenshot alongside.



▼ Library - soundtrack1.fla

soundtrack1.fla

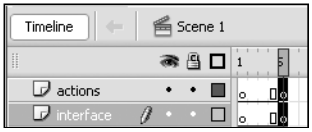
11 items

Name	Type	Use Count	Linkage
beats01.wav	Sound	-	Export: beats01
beats02.wav	Sound	-	Export: beats02
beats03.wav	Sound	-	Export: beats03
beats04.wav	Sound	-	Export: beats04

Depending on the setup of your operating system, the names of the imported sound files may or may not display the `.wav` *filename extension. The values in the* Name *column are unimportant. Just make sure that the values in the* Linkage *column are as shown in the preceding screenshot—without* `.wav` *on the end.*

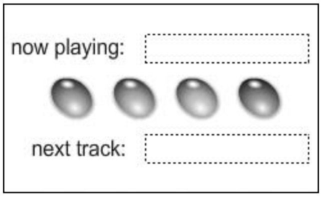
3. Now it's time to set up the main timeline. You need two layers: actions and interface (you should be able to guess what you're going to do with them!). Lock the actions layer and use F6 to insert keyframes on frame 5 of each layer.

   You're going to put **everything** into frame 5 and not into frame 1 as you might have expected. There's a very good reason for this: some versions of Flash Player seem to take a few frames for objects to "settle," and since your movie depends on accurate sound timing, you want to make sure that the player doesn't compromise this accuracy.



Here's a useful rule of thumb: if your movie relies a great deal on the accurate timing of events, it's always a good idea to start adding code a few frames in. This ensures sounds and other objects marked for export in the first frame will completely initialize before you use them in your code.

4. Let's build the interface now. This will consist of four buttons (for selecting sound clips) and two dynamic text fields (to show which sound is currently playing and which one is up next), with a label for each (simple static text fields). Here's how frame 5 of your interface layer might look (if you want to use the same buttons, they come from Window ➤ Common Libraries ➤ Buttons ➤ classic buttons ➤ Ovals).



5. The next step is to name each of the instances you'll want to use in your ActionScript. Begin by naming each button after the sound clip you want it to trigger: beats01_btn, beats02_btn, beats03_btn, and beats04_btn.

6. Give the dynamic text fields the instance names nowPlaying_txt (top text field) and nextTrack_txt (bottom text field).

7. Now that the stage is set, let's write the ActionScript that will bring it to life. All the code needs to go into frame 5 of the actions layer, so select this keyframe and pin the Actions panel to it before you start.

   Here's how it will work. When you click any of the buttons, the sound file being played will change only when the current audio loop has finished, *not* when you click a button. If all your loops are of the same length and have the same musical properties (beats per minute and style), this results in a seamless transition from one loop to the next, allowing you to create an interactive soundtrack of your own choosing.

8. Begin by initializing the soundNext variable, which indicates which track you want to play next.

```
// initialize starting track
var soundNext:String = "beats01";
```

9. Next, create your sound instance, attach the specified sound clip, and tell it to play through once. Also update the nowPlaying_txt text field with the name of the clip.

```
// set up dynamic sound object
var dynamicSound:Sound = new Sound();
dynamicSound.attachSound(soundNext);
dynamicSound.start(0, 1);
nowPlaying_txt.text = soundNext;
```

10. Now define the function that selects the next sound, and assign it as the onRelease event handler for each of the buttons. Add this code *above* the existing script:

```
// define handler for button events
function selectNextSound():Void {
  soundNext = substring(this._name, 0, 7);
  dynamicSound.attachSound(soundNext);
  dynamicSound.start(0, 1);
  nowPlaying_txt.text = soundNext;
}
beats01_btn.onRelease = selectNextSound;
beats02_btn.onRelease = selectNextSound;
beats03_btn.onRelease = selectNextSound;
beats04_btn.onRelease = selectNextSound;
```

   Using a named function like this is a lot easier than writing out four identical anonymous functions as event handlers, one for each button. Let's go through the function to see what it does.

```
soundNext = substring(this._name, 0, 7);
```

As you know, the `this` keyword refers to whatever generates the event that calls a function. For instance, when the function is called because the `beats02_btn` button generates an `onRelease` event, then `this` refers to the `beats02_btn` button. So, `this._name` gives you the instance name of the `beats02_btn` button. However, you need to extract the name of the sound instance it controls.

You do this by passing the button's instance name to the `substring()` method of the `String` class. This takes three arguments: the string that you want to extract a substring from, where to start, and the number of characters to extract. As with arrays, you count from 0 in a string, so the second argument says "Begin at the first character," and the final argument says "Extract seven characters." This gives you `beats02`, which can then be passed to `attachSound()` like this:

```
  dynamicSound.attachSound(soundNext);
  dynamicSound.start(0, 1);
```

You also update the top text field to signify which sound you've just requested:

```
  nowPlaying_txt.text = soundNext;
```

So clicking any of the four buttons makes `soundNext` equal to one of four string values. These string values are the same as the four linkage names for our sound clips, so you can safely feed `soundNext` into the `attachSound()` method on your sound instance.

11. Finally, add `stop()` to the end of the script so far, so you don't go looping around the five-frame timeline:

```
// stop the timeline
stop();
```

The script so far should look like this:

```
// define handler for button events
function selectNextSound():Void {
  soundNext = substring(this._name, 0, 7);
  dynamicSound.attachSound(soundNext);
  dynamicSound.start(0, 1);
  nowPlaying_txt.text = soundNext;
}
beats01_btn.onRelease = selectNextSound;
beats02_btn.onRelease = selectNextSound;
beats03_btn.onRelease = selectNextSound;
beats04_btn.onRelease = selectNextSound;
// initialize starting track
var soundNext:String = "beats01";
// set up dynamic sound object
var dynamicSound:Sound = new Sound();
dynamicSound.attachSound(soundNext);
dynamicSound.start(0, 1);
nowPlaying_txt.text = soundNext;
// stop the timeline
stop();
```

12. Test the movie for a taste of what can happen when Flash sound goes wrong. Each sound is **musically unsynchronized**. It plays when you click a button and the overall result is one sound piling on top of the next.

> *It may take longer than usual for the movie to compile. This is because the WAV files come to more than 1MB. The resulting SWF will be much smaller.*

Restoring order to the chaos

1. The whole point of this exercise is to put the onSoundComplete event through its paces, so let's do that. Change the button event handler as shown here (the final code is in soundtrack2.fla):

```
function selectNextSound():Void {
  soundNext = substring(this._name, 0, 7);
  nextTrack_txt.text = soundNext;
}
```

   This time, the buttons don't change the sound immediately; they only set the soundNext variable to the next sound file that has been requested.

2. What you want to do is change to the requested sound "on the beat," or when the current sound has completed. You guessed it—onSoundComplete. Add the following code just above the stop() command like this:

```
nowPlaying_txt.text = soundNext;
// trigger next sound
dynamicSound.onSoundComplete = function() {
  dynamicSound.attachSound(soundNext);
  dynamicSound.start(0, 1);
  nowPlaying_txt.text = soundNext;
};
// stop the timeline
stop();
```

3. Test run the movie again, and you'll hear quite a difference! When you listen to it, you'll understand why it's so ingenious. The sounds won't just change as soon as you click one of the buttons—they'll change only after the track that's currently playing has finished. This means that your little techno tune stays in a musical beat sequence no matter what you do!

   When you first click one of the buttons, you'll see the nextTrack_txt text field updated immediately, but there's no change to the nowPlaying_txt field and, more important, there's no change in the sound. However, once the current clip has played through, both will change to match the clip shown in nextTrack_txt.

> *This is sequencing to the nearest beat in a four-beat sequence. A dance DJ or remix artist would call this **mixing in on the beat**, and most other musical types would call it a **sequencing loop**.*

4. Navigate to the folder where you are building your test files. Take a look at the size of `soundtrack2.swf` and of the WAV files you used to build it. Although the original sound files come to more than 1MB, the SWF itself is a mere 55KB! Naturally, the sound quality is not as clear as the original files, but the degradation is minimal. The download files also contain a version created using MP3 files, `soundtrack_mp3.fla`. Ironically, the resulting SWF is four times the size of the one created using WAV files, and it suffers from tiny gaps between the consecutive tracks. We'll come back to the question of file size later in the chapter.

Using ActionScript to control volume and balance

Thanks to the `Sound` class, you can use ActionScript to manipulate your sound clips in subtler ways than you've looked at so far (where all you've really done is control the start and stop instant). In particular, you can alter two aspects of the original sound: the **volume** (how loud the sound is) and the **panning** (in ActionScript, this is the balance of sound you hear in each speaker).

Changing the volume of a sound The `setVolume()` method of the `Sound` class allows you to vary the total volume produced by a sound object. The values that change this are pretty obvious:

- `mySound.setVolume(0)`: This value gives you 0% volume (silence).
- `mySound.setVolume(100)`: This value gives you the maximum volume, which largely depends on your speaker setup.

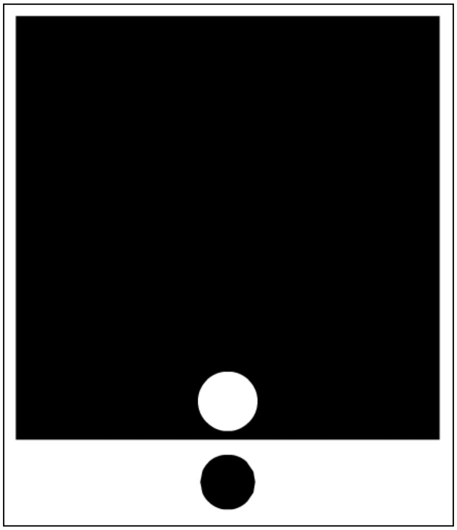
Since the acceptable values run between 0 and 100, the number represents the percentage of maximum volume.

Changing the sound balance The `setPan()` method of the `Sound` class allows you to vary the sound level in each speaker—that is, the balance between them. The name of the method comes from the idea of panning a camera to follow the movement of a subject. The values that you can apply to it alter the sound accordingly:

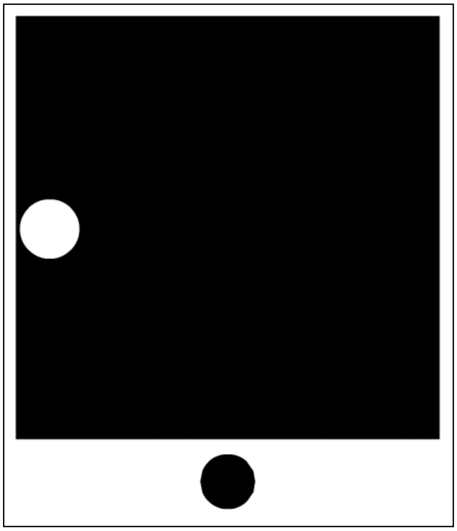
- `mySound.setPan(0)`: This value gives an equal balance in the left and right speakers.
- `mySound.setPan(-100)`: This value gives the full current volume in the left speaker, and nothing in the right.
- `mySound.setPan(100)`: This value gives the full current volume in the right speaker, and nothing in the left.

Let's put this information to use and design a simple sound control system.

When you look at the following images, imagine the black square is a stage as viewed from above. The black dot is you sitting in front of the stage, and the white dot is a musician on the stage. As you drag the white dot left and right on the stage area, you want ActionScript to alter the stereo image dynamically by panning and to change the volume of the music to reflect the performer's movement backward and forward on the stage.



Say the musician is standing right in front of you by the footlights. You'll hear the sound at a maximum volume, in your left and right ears equally. If the musician then moves toward the back of the stage and plays at the same level, the sound will become equally faint in both of your ears—assuming, of course, there's no amplification.



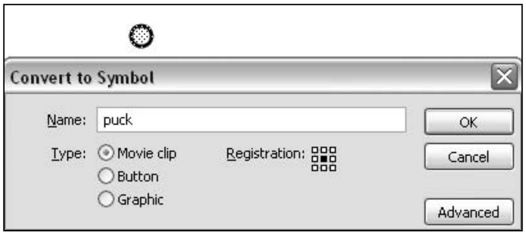
If the musician moves away from you and across to your left, the sound will become fainter, and you'll hear more sound in your left ear than in your right ear (and vice versa if the musician moves closer and to your right).

Building sound controls

You'll reproduce this setup, creating a draggable white dot. This dot will control a sound instance via the `setVolume()` and `setPan()` methods. The effect of this will be that the sound you hear from your computer (assuming you have stereo speakers appropriately connected and placed) will be exactly the same as if you stood in the same position as the black dot in the preceding description. This example is called `controls.fla` in the download for this chapter if you get stuck.

1. Open a new Flash document and set up two layers called actions and interface. Lock the actions layer to avoid adding any graphics to it by accident. Use File ➤ Import ➤ Import to Library to import any of the loop files you've used before. Right-click/Ctrl-click to select Linkage from the context menu and set the sound clip to Export for ActionScript with a linkage identifier of loop.

2. Draw a circle about 15 pixels in diameter with a white fill and black stroke (the exact size is unimportant). Convert it to a movie clip called puck by pressing F8, and set the registration point to the center as shown in the screenshot.



Convert to Symbol

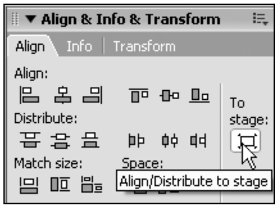
Name: puck

Type: ● Movie clip ○ Button ○ Graphic

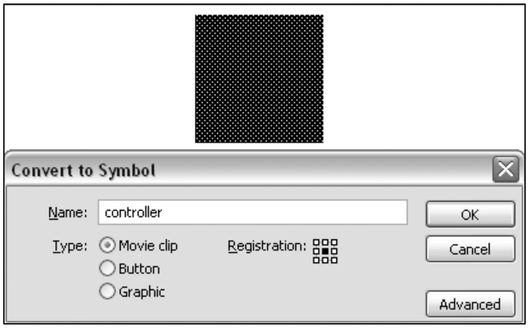
Registration: center

3. Delete the instance of the puck from the stage. The movie clip symbol remains in the Library, ready for use a little later.

4. Create a strokeless black square, 100 pixels across. Use the Align panel (Window ➤ Align) to align the square horizontally and vertically in the center of the stage.



5. With the square still selected, press F8 to convert it into a movie clip called controller. Set the registration point in the center, as shown in the screenshot.



Convert to Symbol

Name: controller

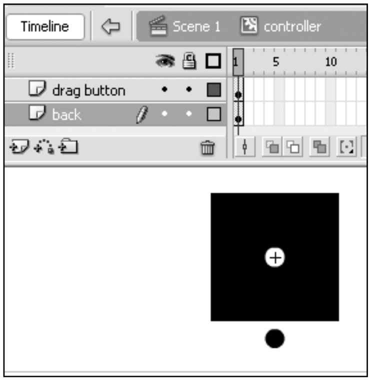
Type: ● Movie clip ○ Button ○ Graphic

Registration: center

6. Double-click the `controller` movie clip to edit it in place. Rename Layer 1 as back, and add a new layer called drag button.

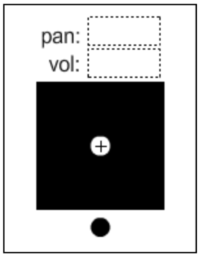
7. Drag an instance of the puck movie clip onto layer drag button, and use the Align panel to align it horizontally and vertically in relation to the stage. In the Property inspector, give it the instance name puck. The puck movie clip should now be in the exact center of the black square.

8. Select layer back and add a small black circle just below the black square (to indicate where you're listening from at the front of the stage), and use the Align panel again to align the horizontal center to the stage. The controller movie clip that you're editing should now look like the screenshot shown alongside.



9. Add another layer to the timeline and call it text. Add a pair of dynamic text fields, giving them the instance names pan and vol. Use static text to the left of each dynamic text field to indicate what it will display.

   The controller movie clip should look like the screenshot alongside when you're done.



10. Click Scene 1 to exit the edit screen. Give the controller movie clip the instance name controller.

11. Select frame 1 in the actions layer and pin the Actions panel to it.

12. You want to make the percussion sound start playing and loop continuously. Add the following code in the Actions panel:

```
var loopSound:Sound = new Sound();
loopSound.attachSound("loop");
loopSound.start(0, 1000);
```

    If you test the movie now, you'll hear the sound playing, but you'll have no control over it. Let's fix that.

13. The next step is to wire up the puck movie clip so that you can use the mouse to drag it about. Amend the code like this:

```
function drag() {
  this.startDrag(false, -50, -50, 50, 50);
  this.onMouseMove = updateAfterEvent;
}
function drop() {
  this.stopDrag();
  delete this.onMouseMove;
}
var loopSound:Sound = new Sound();
loopSound.attachSound("loop");
loopSound.start(0, 1000);
controller.puck.onPress = drag;
controller.puck.onRelease = drop;
controller.puck.onReleaseOutside = drop;
```

> *Notice that you use dot notation here to refer to the movie clip* puck *inside the* controller *movie clip, which is on the main timeline, hence the dot notation path* controller.puck.

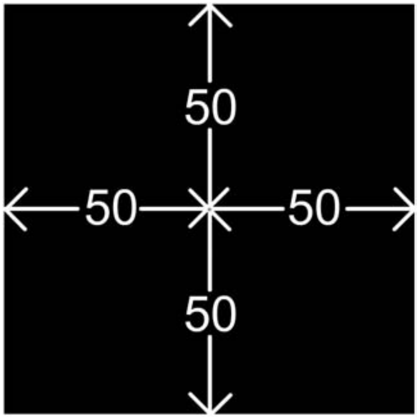
Although there's a lot of new code, it isn't that hard to see what's going on. The function drag() is used to define what happens when the puck is click-dragged, and drop() defines what happens when the puck is dropped.

The function drag() constrains the movement of the puck movie clip by adding five arguments to the startDrag() method like this:

```
this.startDrag(false, -50, -50, 50, 50);
```

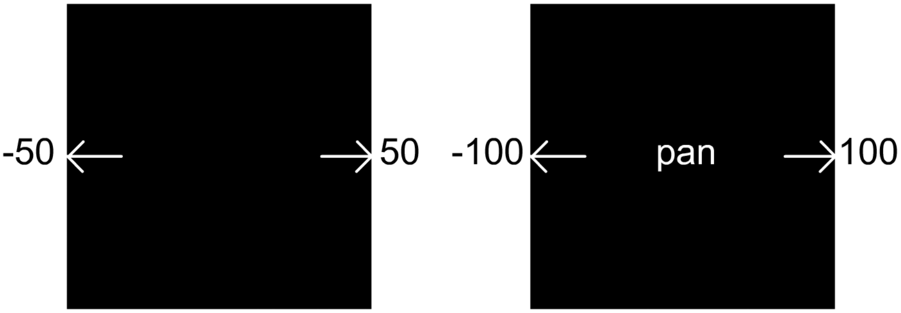
The first argument must be either true or false. If true, it specifies that the movie clip being dragged is locked to the center of the mouse pointer. If false, it's locked to the point where the movie clip was first clicked. The remaining four arguments specify how many pixels the movie clip can be dragged in each direction, listed in the following order: left, top, right, bottom.

Since you made the black square of the controller movie clip 100 × 100 pixels and placed the puck movie clip in the exact center, you want it to move no more than 50 pixels in any direction, as shown in the diagram. Remember that in Flash *down is positive*, so the third argument (which controls how far the movie clip can be dragged toward the top) is *minus* 50.



14. OK, the puck movie clip now works properly, but it makes no difference to the sound you hear. Nor does it give you values for volume and pan in the dynamic text fields. You need to evaluate `setVolume()` and `setPan()` values for the sound object. The puck movie clip can be dragged 50 pixels to the left and 50 pixels to the right from its initial starting position. You need to scale this to –100 to +100 for use in the `setPan()` method. Since the x-coordinate of the puck ranges from –50 to +50, all you need to do is to double the figure.

pan level = (puck x-coordinate) x 2

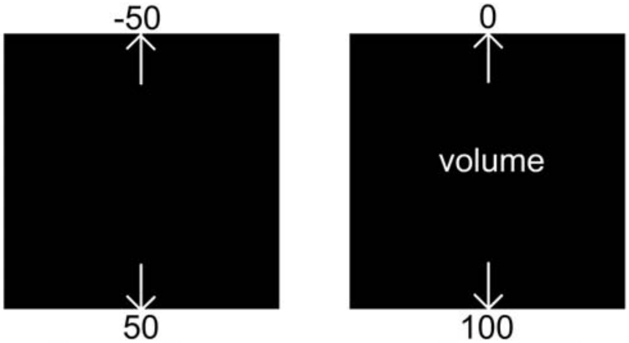


Likewise, you can drag the puck 50 pixels up and 50 pixels down from its starting position. This must be translated to a range of 100% to 0 for use with the `setVolume()` method. To do this, you just need to add 50 to the button's y-coordinate:

(100 to 0) = (50 to -50) + 50

That is,

volume level = (puck y-coordinate) + 50



15. The final code adds an `onEnterFrame` event to the puck every time you click-drag it. This new event converts the puck movie clip's position into two variables (pan and `vol`), which are used to control the panning and volume of the sound instance `loopSound`. You also update the two text fields to show the current pan and volume values. Here's the finished code:

```
function drag() {
  this.startDrag(false, -50, -50, 50, 50);
  this.onEnterFrame = function() {
    pan = 2*this._x;
    vol = 50+this._y;
    loopSound.setPan(pan);
    loopSound.setVolume(vol);
    this._parent.pan.text = pan;
    this._parent.vol.text = vol;
  };
  this.onMouseMove = updateAfterEvent;
}
function drop() {
  this.stopDrag();
  delete this.onEnterFrame;
  delete this.onMouseMove;
}
var pan:Number = 0;
var vol:Number = 0;
var loopSound:Sound = new Sound();
loopSound.attachSound("loop");
loopSound.start(0, 1000);
controller.puck.onPress = drag;
controller.puck.onRelease = drop;
controller.puck.onReleaseOutside = drop;
```

16. Try the movie once again. Drag the puck around the stage and you'll hear the sound move from side to side, rising and falling in volume as you move the object up and down. The values of pan and volume are updated to reflect the changes.

Also worth noting is the way you're dynamically creating events in the code:

- Rather than allow the `onEnterFrame` to run all the time, you delete it in the `drop()` function, so it runs only as long as the puck is being dragged.
- The puck is redrawn every time you move the mouse, which is much faster than the frame rate. The text fields are updated at the slower frame rate. This makes for a much more efficient and responsive SWF, as Flash spends more time creating a smoothly animated puck. Users won't notice that the sound methods and text fields are being controlled at a much slower rate, and they'll get the impression that the whole thing is silky smooth. This is one way to overcome the relative slow speed of Flash Player—cheat!

The sound variations are occurring in real time, in response to how you drag the puck. This is a template for creating a dynamic soundscape for your websites or Flash games. As the sound source moves around the screen, you can create the appropriate sound effects on the fly with the Sound *class and ActionScript.*

Dealing with large sound files

As you've probably noticed by now, most sound files tend to weigh in on the heavy side. A lot of information goes into making a small snippet of sound, and the size of your audio files will most likely reflect this. Although broadband Internet access is becoming widespread, multimegabyte downloads won't make you very popular with most people. This is one reason sound on the Web hasn't been as common in the past as it might have been.

In recent years, though, two technologies have helped change things for the better: **compression** and **external sound files**.

Using compression to reduce download times

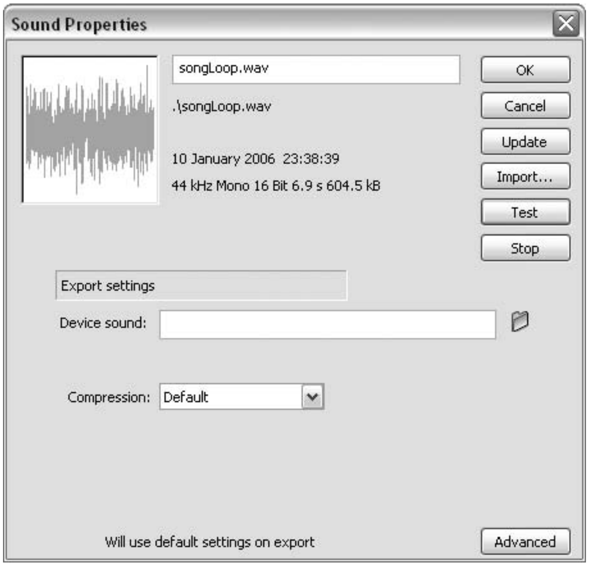
As mentioned before, by default Flash uses MP3 to compress your sound files when it compiles the SWF. Many developers leave Flash to do it automatically, but you also have the option of controlling the compression for individual sound files.

Here are some points to keep in mind when creating Flash files that include sound:

- Flash has the option to compress the whole SWF (File ➤ Publish Settings ➤ Flash). If your content is already compressed, then you should consider not compressing it further.
- You should consider using sound files that compress well. Generally, low, bassy sounds compress much better than high-frequency sounds.
- Rather than accept the default compression settings, you should look at each sound file separately and find the best compression settings for each. Sound is bandwidth heavy, and it can be the biggest part of your SWF file, so take every precaution to minimize the file sizes.

Checking the compression settings of a sound

1. Open `controls.fla`, the file you used in the last exercise, if it's not already open. Right-click/Ctrl-click songLoop in the Library panel (it may be displayed as songLoop.wav), and select Properties from the context menu. The Sound Properties dialog box will open, as shown in the screenshot.



Sound Properties

songLoop.wav

.\songLoop.wav

10 January 2006 23:38:39

44 kHz Mono 16 Bit 6.9 s 604.5 kB

Export settings

Device sound:

Compression: Default

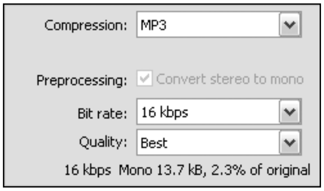
Will use default settings on export

2. The Compression drop-down menu should be set to Default, which means that Flash will use its own default settings. Unless you changed them in the Publish Settings dialog box, the default settings are MP3 at 16 kbps in mono.

3. Click the Test button. This plays the original sound file without any compression.

4. Change the setting of the Compression drop-down menu as shown in the screenshot alongside, and click Test again. This is how it will sound when the movie is compiled. It's not as sharp as the original, but it's acceptable for most purposes.

   You can also see the dramatic effect the change has on file size. The line of text at the bottom of the dialog box compares the original file size with the compressed one.



Compression: MP3

Preprocessing: ☑ Convert stereo to mono

Bit rate: 16 kbps

Quality: Best

16 kbps Mono 13.7 kB, 2.3% of original

Note that the Quality *setting has no effect on file size. It simply changes the trade-off between compressed sound quality and the amount of time it takes to turn your FLA's sound files into compressed SWF assets. For most modern computers, consider setting this to* Best*, unless doing so makes SWF compilation unacceptably slow.*

5. Experiment with different settings to compare file sizes and sound quality. In addition to MP3, other compression formats available are ADPCM (Adaptive Differential Pulse Code Modulation), Raw, and Speech. ADPCM is used only for short sounds like button clicks. Raw plays more quickly than MP3, but produces much bigger files, so it is really suitable only for applications that will be played directly from a hard drive. Speech is self-explanatory.

If you have a movie that makes a lot of use of sound, you should go through each sound's properties and decide which can be compressed without sacrificing too much quality. Any compression settings made in the Sound Properties dialog box override the Flash default.

Loading sound from external files

Flash allows you to load sounds as individual files, separate from the SWF that actually plays them. The advantages of this include the following:

- Your sound files don't increase the size of the SWF, thereby reducing the initial download.
- You can load sound files only when they're requested. For example, if you're developing a Flash jukebox that allows the user to select from a number of different song files, you don't want to load the songs until they're selected to be played. To do otherwise may involve SWFs that run into MBs!
- You can choose which versions of a sound file to load at runtime. You can select between high-quality and low-quality files to cater to users utilizing different bandwidths.

Flash can load a sound file in two ways: it can load a streaming sound or an event sound.

- A **streaming sound** is just that: it streams from the Web. It's downloaded from the Web every time you want to play the associated audio, and the associated audio can begin before the sound is fully loaded.
- An **event sound** is a sound you want to start playing as soon as something happens, such as a button click. An event sound needs to load in fully before it can start playing. Once the event sound has loaded, though, you can use it much like a sound in the Library; you don't have to reload it every time you want to start the sound again.

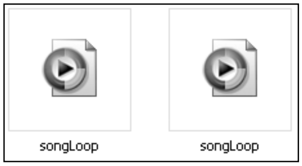
Generally, you should use streaming sounds for long pieces of audio, such as songs for a jukebox. Event sounds are more useful for spot effects, such as UI click sounds.

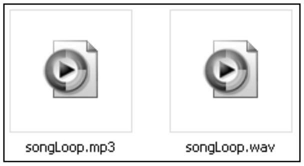
A streaming sound will start playing as soon as enough of it has loaded in to fill a sound buffer, which is five seconds by default. You can change the buffer duration by changing the global property `_soundbuftime`. (Note that this will change the buffer time of the whole Flash Player, not just that of the current movie clip.) Once the buffer has filled to the minimum level, the sound will continue to load the sound file, playing it at the same time. Once the sound has completed, you need to load it again to hear it again.

Many people seem to have trouble with loading external sound files, so here are two basic things to look out for:

- MP3 is the only format you can use when loading external sound files into Flash. Older versions of Flash Player are very picky about exactly which MP3 files they will load. This problem has now basically disappeared with Flash Player 8. If your target audience is likely to be using an older version, you should test your files carefully. You can get older versions of Flash Player for testing purposes from `www.macromedia.com/go/tn_14266`.

- Another issue to watch out for is that your operating system may show MP3 files and WAV files using the same file icon, so be sure you're asking Flash to load an MP3! The default setting on Windows systems is to hide the filename extensions of known file types. As a result, you may see something like the screenshot on the left, even though one file is an MP3 file and the other is in WAV format. Life is a lot less confusing if you turn on all filename extensions in Windows Explorer (Tools ➤ Folder Options ➤ View ➤ Advanced Options). In Mac OS X, you can turn on the display of filename extensions in Finder by accessing Finder ➤ Preferences ➤ Advanced. Alternatively, you can display filenames selectively for individual files or groups of files by highlighting them, choosing File ➤ Get Info, and deselecting Hide extension.





Let's have a look at using both streaming and event sounds. For the streaming sound, we got David to rummage around in his sound archive, and he came up with something of a rarity. One day, back in the early 1990s, he was just about to get on a train in suburban Tokyo when he heard some strange music drifting across from the local shopping street. It turned out to be a group of gaudily dressed wandering minstrels known as **chindonya** (because most of their music sounds like "chin-don"). They were advertising a grand sale to mark the opening of Dai-ichi Stoa ("Number One Store"), which—in spite of its impressive name—was a tiny neighborhood shop. In years gone by, *chindonya* were a common sight and sound on the streets of Japanese cities, hired by owners of small shops to drum up business—literally. Now, very few are left. Although David didn't have a camera with him, he did have a tape recorder, and you can hear the results in this next exercise.

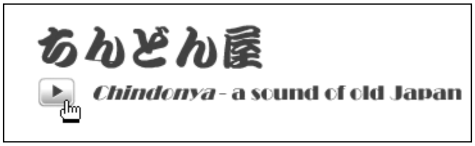
> *If you're curious about* chindonya *and want to learn more, take a look at* `www.u-stage.com/chindonyaE.html`.

Loading a streaming sound

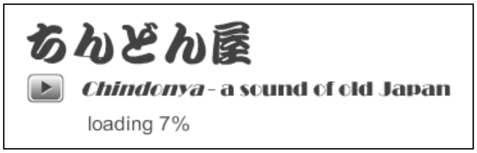
You'll need the files `soundLoader.fla` and `chindonya.mp3` from the download for this chapter. For the effect to work properly, you need to view the finished SWF over the Internet. Since the sound file is nearly 2MB, you may prefer to view it at the friends of ED website, rather than upload it to your own website. You can find the link on this book's page at `www.friendsofed.com/book.html?isbn=1590596188`.

1. Publish `soundLoader.fla` (File ➤ Publish).
2. If you have a fast connection, copy the files created, `soundLoader.swf` and `soundLoader.html`, to your remote server. Copy the file `chindonya.mp3` to the same location.
3. Browse to `soundLoader.html` to view the SWF over the Web. You can also view the files locally, but you won't see the full effect.

When you view the page, you'll see something like the following screenshot:

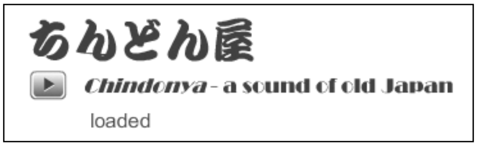


Clicking the button will make a loading message appear. Once the percent loaded reaches a certain value (around 5%), you'll begin to hear the song play over the Web. The percentage loaded message will continue to advance as you listen to the sounds of yesteryear.



loading 7%

Once the sound file has fully loaded (a second or so after you see the loading value reach 100%), you'll see the message change to tell you the file has finished loading. You'll most likely continue to hear the *chindonya* for some time after this message has appeared. If you run the SWF from your local hard drive, the fully loaded message (shown in the following image) will appear immediately.



loaded

The thing to notice here is that the MP3 file is nearly 2MB, but you don't have to wait for 2MB worth of downloaded SWF before you see the button, and you don't have to wait for the full 2MB to load before you hear something.

If you listen to the song over a 56K or lower bandwidth connection, you may hear the sound file occasionally pause. This is because the MP3 is loading at a slower rate than that needed to play it without any pauses. Add the following line at the top of the code in the Actions *panel in* `soundLoader.fla`*:*

```
_soundbuftime = 10;
```

This increases the sound buffer to ten seconds. Adjust the length of the buffer as necessary.

This is the basis of a simple web music player using streaming sound!

Let's have a look at how it works. Open `soundLoader.fla` and have a look at the main timeline. This consists of two layers, actions and interface. Layer interface contains a button, `startLoad`, plus a dynamic text field, `loaded`, and associated static text.

The main meat is, of course, in the code on layer actions:

```
// Display how much of sound file has loaded
function loadProgress():Void {
  loaded.text = "loading: " + Math.round(100*song.getBytesLoaded()/ ➥
song.getBytesTotal())+"%";
}
// Function to be run once sound is fully loaded
function loadDone():Void {
  loaded.text = "loaded";
  delete _root.onEnterFrame;
}
// Create sound instance and assign function to onLoad event
var song:Sound = new Sound();
song.onLoad = loadDone;
// Button onRelease event handler
startLoad.onRelease = function() {
  song.loadSound("chindonya.mp3", true);
  _root.onEnterFrame = loadProgress;
};
```

There are two functions: loadProgress() and loadDone(). The first uses two methods of the Sound class, getBytesLoaded() and getBytesTotal(). The method names are self-explanatory, and they're used in a simple percentage calculation to work out how much of the sound file has been loaded. If you look at the last line of the script, you'll see that loadProgress() is assigned to the onEnterFrame event of the main timeline (_root). So it updates the dynamic text field on every frame.

The second function, loadDone(), is designed to run when the sound file is fully loaded into the SWF. As well as updating the text field with the "loaded" message, it deletes the onEnterFrame event handler.

The main code then sets up the sound instance and creates an onRelease script for the button. This runs the line to load the MP3:

```
song.loadSound("chindonya.mp3", true);
```

The loadSound() method has two arguments: the URL of the MP3 file you want to load and whether you want to stream the sound file. For a streaming file, the second argument is true. If you set the second argument to false, the sound won't play until it's fully loaded—in other words, it becomes an event sound, which we'll look at next.

Sometimes you may want to stop a loadSound()*. There is no method to do this. The best way to handle this is to delete the sound instance you're loading into. In this example, you would include the following line of code within a "stop download button"* onRelease *script:*

```
delete song;
```

Loading an event sound

Loading your event sounds from external files, rather than placing them into a Library, has several advantages:

- You can decide when to load the sounds. Storing them in the Library means that you have to load all the sounds at frame 1 of your site, which can cause an unacceptably long delay before your movie starts running.
- Your SWF becomes much smaller, allowing it to load much faster.
- You can choose which set of sounds you load. For example, you might want different sets of sounds for low-bandwidth users and high-bandwidth users. When you're creating an international site, you can choose different languages.
- Keeping the sound files separate from the SWF means that you (or even the client) can modify the site easily. You simply replace the sound files—no SWF redesign necessary!

To load an event sound, set the second argument of the `loadSound()` method to `false`, as shown in the following listing:

```
var clickSound:Sound = new Sound();
clickSound.onLoad = function() {
  this.start(0, 1);
};
clickSound.loadSound("beats01.mp3", false);
```

The only downside of using loaded event sounds is that any attempt to use `start()` before the `onLoad` event has been triggered will fail. You must also define the `onLoad` event handler *before* you attempt to use the `loadSound()`. If the `clickSound.loadSound` line in the preceding code appeared before the `clickSound.onLoad` definition, the code would not work online (although you would probably get away with it offline).

If you want to check whether a sound file has actually loaded (for both event and streaming sound), you can add a parameter called `success` to the `onLoad` event handler. To modify the preceding code so that you can detect a failure of the sound file to load, use something along the lines of the following code:

```
var clickSound:Sound = new Sound();
clickSound.onLoad = function(success:Boolean) {
  if (success) {
    this.start(0, 1);
  } else {
    // code to handle failure to load the sound
  }
};
clickSound.loadSound("beats01.mp3", false);
```

A final point to note when you use separate MP3 files is the issue of *security*. When you embed MP3 files into the SWF, the user can't easily use the MP3 files for his or her own use (because you can't listen to a SWF on an iPod!).

Many recording artists shy away from making MP3 files of their records available on a traditional website, because the listener might just burn the MP3 onto a CD-ROM, make it available on file-sharing networks, or save it onto an iPod. Flash allows a much greater level of protection when the MP3 is embedded within a SWF (although it can still be extracted using the right tools and a bit of know-how).

When you load MP3 files separately, they become accessible (the user will be able to find the MP3 files in the browser cache). To avoid this, you can rename the MP3 files in one of the following two ways:

- Leave the `.mp3` extension off the end of the filename. So instead of this:

  ```
  clickSound.loadSound("beats01.mp3", false);
  ```

  use this:

  ```
  clickSound.loadSound("beats01", false);
  ```

  and place a file called `beats01` (instead of `beats01.mp3`) on your web server. This prevents anyone from playing the file on an iPod unless the `.mp3` extension is added back to the filename.
- Give your MP3 files a filename extension other than `.mp3`. Renaming your MP3 file to `top_bar.gif`, for example, would confuse most casual hackers.

  ```
  mySound.loadSound("top_bar.gif", false);
  ```

  This will work perfectly if `top_bar.gif` is actually an MP3 file, because Flash ignores filename extensions. It will use the file's internal "header" information to determine what kind of file it really is. Flash needs to see extensions as part of the filename string, but it doesn't actually recognize them in any other way.

In all honesty, though, neither of these methods will deter a determined hacker. Valuable files should be protected by a more sophisticated security system or not put on the Internet in the first place. A sensible alternative is to offer visitors just a taster—say, 30 seconds of music—enough to make them want more, but not enough to rip off.

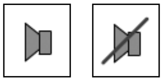
Silence can also be golden

Sound brings the Web alive, but it can also kill stone dead someone's interest in your site. In the early days of the Internet, blinking text, twirling icons, and snazzy bits of eye candy were a great relief from blocky text on a dull gray background. However, something that starts out as eye-catching and cool can quickly become intensely irritating. Using sound in your websites runs the risk of alienating many, if not all, of your visitors. That quirky "boing" or "kaboom" may be just the right thing for an online game, but is it the right thing for a business site? What amuses the first couple of times can induce homicidal tendencies in even the most even-tempered visitors when the novelty wears off.

That doesn't mean you shouldn't use sound—just that you should use it wisely, and always give your visitors the opportunity to turn off sound effects. It's easy enough. You've already learned that using stop() on any sound instance in your movie will halt all sounds, as long as you don't pass an argument to it. All you need to do is add a button that lets users choose to turn off the sound. If you have just a single sound instance, it's as simple as this (you'll find this example in soundOff.fla):

```
toggleSound_mc.onRelease = function() {
  if (playing) {
    mySound.stop();
    this.gotoAndStop("soundOff");
    playing = false;
  } else {
    mySound.start(0, 1000);
    this.gotoAndStop("soundOn");
    playing = true;
  }
};
var mySound:Sound = new Sound();
mySound.attachSound("songLoop");
mySound.start(0, 1000);
var playing:Boolean = true;
```

This will run songLoop when the movie first starts, but the sound can be toggled on and off by clicking a movie clip called toggleSound_mc, which uses the common conventions of a speaker to indicate that sound is being played and a speaker with a diagonal line through it to indicate that sound has been turned off. The Boolean variable, playing, keeps track of whether the sound is being played.



Playing sound by default at least lets your visitors know that you're sound-enabled, but you should always consider whether it might not be a better idea to let your visitors opt *in* to sound rather than forcing them to opt out. If they don't like your music, they may shut down their browser window so fast, they never get a chance to see how cool your site really is. On the other hand, if they don't share your taste in music . . .

There are no simple answers. If you know your target audience really well, you'll probably give them what they want. That said, it's always a good idea to give visitors the option to kill the sound, even if only temporarily.

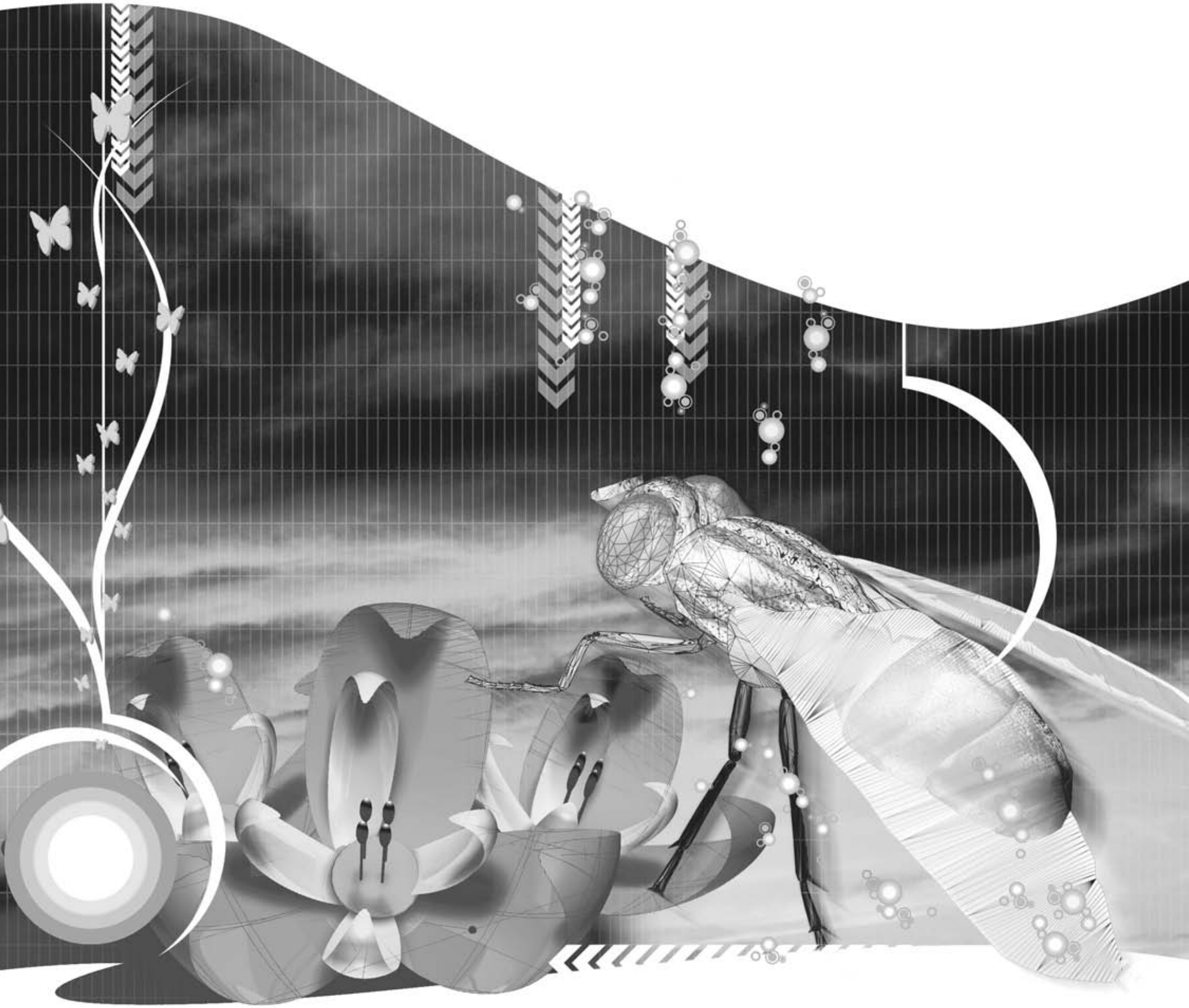
Summary

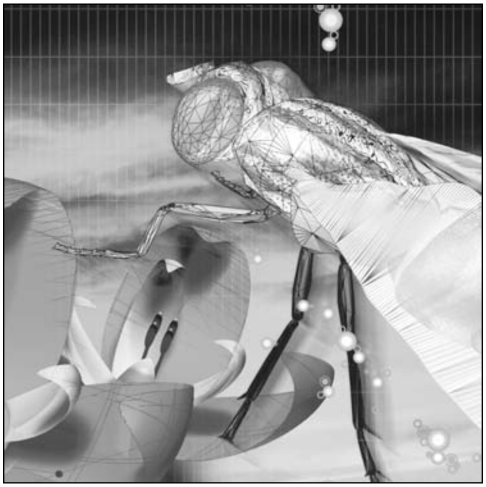
This chapter has taught you something that a lot of people seem to be struggling with: how to tame the `Sound` class. You can use the time you've saved and devote it to building your own Academy Award–winning soundtracks. With 32 total channels of audio now instead of 8, we expect to hear a lot more audio on the Net!

In this chapter you learned

- How to create a new sound object and control it with ActionScript
- How to add pan and volume controls
- How to stream audio and load sounds dynamically

In the next chapter, we'll look at another important aspect of importing external data into Flash using the `XML` class. You'll build the data structure for the Futuremedia site with a simple XML file, and then import it into the site, finally giving it the content that's been missing so far. By storing the content as XML, you give the site great flexibility.





Chapter 13

LOADING DYNAMIC DATA WITH XML

What we'll cover in this chapter:

- Understanding how XML pages are structured
- Using the XML and XMLNode classes to load data into Flash
- Turning an XML object into something Flash can use
- Populating the Futuremedia site with data

Unless you've been on Mars for the past few years, you're bound to have heard a lot about Extensible Markup Language (XML) and how important it is to the Web. It's a buzzword that really buzzes. Yet, in spite of all the noise, many web designers rarely come into contact with XML, mainly because you can't build websites with XML alone—at least not with most XML documents. XML is surrounded by mystique. You know it's cool, but you're not quite sure how.

The great news is that XML is incredibly simple. The not-so-great news is that working with XML isn't always easy. The difficulty stems from the fact that XML documents are designed to be structured and predictable. Although that *should* make working with XML relatively straightforward, the problem is that XML demands that you follow very strict rules. Get one wrong, and everything falls apart. It also requires a logical and methodical approach, something that doesn't always come naturally to designers and artistic types used to working in a visual medium, such as Flash.

Still, if you have progressed this far in the book, you'll know that ActionScript can be very fussy, too. Once you understand the basic principles behind XML, it's not all that hard. By the end of the chapter, we hope to have dispelled some of the mystique by showing you just how powerful Flash can be in combination with XML. What you're going to do is build an XML document that reflects the structure of the Futuremedia site, and then use two ActionScript XML classes to populate the data structure that you built in Chapter 11.

First of all, though, if you need to be brought up to speed on what XML is, the following section presents a quick introduction. Even if you're familiar with XML, make sure you understand the concept of nodes in the section titled "How an XML document is structured," as this is essential to working with XML in Flash.

XML 101

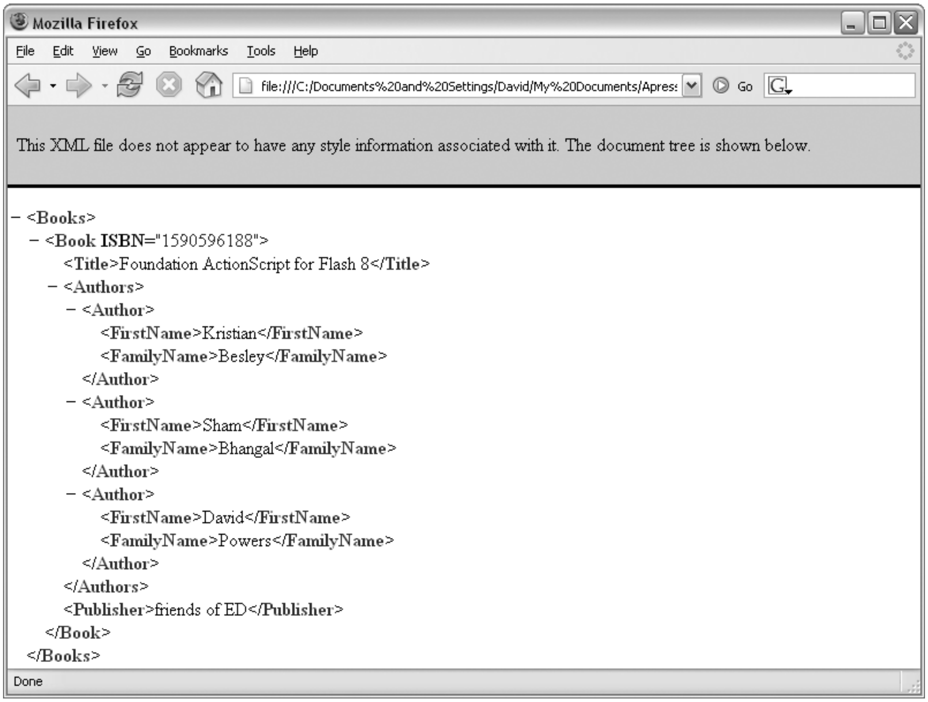
XML is closely related to Hypertext Markup Language (HTML), the basic building block of all web pages and, although you may not realize it, version 1.0 of Extensible Hypertext Markup Language (XHTML) is HTML 4.01 rewritten according to the rules of XML. It looks reassuringly familiar to web designers, but XHTML is only one form of XML.

- XHTML has a fixed range of tags and attributes. To use the XML terminology, it has its own **vocabulary**.
- The XHTML vocabulary is concerned solely with the structure of a web page (`<head>`, `<body>`, `<p>`, `<table>`, and so on).
- XML has other specialized vocabularies, such as Mathematical Markup Language (MathML), which allows you to embed equations into web pages and other documents.
- XML allows you to create your own vocabulary. In other words, you can create your own tags. More often than not, tags are used to describe the data they contain (e.g., the following example uses `<Book>` to store details of a book).

The following is a simple example of an XML document (`book.xml` in the download files for this chapter) that contains details about this book:

```
<?xml version="1.0" ?>
<Books>
  <Book ISBN="1590596188">
    <Title>Foundation ActionScript for Flash 8</Title>
    <Authors>
      <Author>
        <FirstName>Kristian</FirstName>
        <FamilyName>Besley</FamilyName>
      </Author>
      <Author>
        <FirstName>Sham</FirstName>
        <FamilyName>Bhangal</FamilyName>
      </Author>
      <Author>
        <FirstName>David</FirstName>
        <FamilyName>Powers</FamilyName>
      </Author>
    </Authors>
    <Publisher>friends of ED</Publisher>
  </Book>
</Books>
```

If you open book.xml in a browser, it will probably look almost identical to the original code. The following screenshot shows it in Firefox.

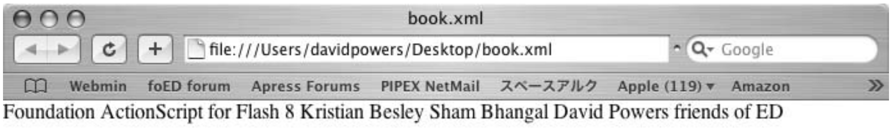


file:///C:/Documents%20and%20Settings/David/My%20Documents/Apress

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
– <Books>
  – <Book ISBN="1590596188">
      <Title>Foundation ActionScript for Flash 8</Title>
    – <Authors>
      – <Author>
          <FirstName>Kristian</FirstName>
          <FamilyName>Besley</FamilyName>
        </Author>
      – <Author>
          <FirstName>Sham</FirstName>
          <FamilyName>Bhangal</FamilyName>
        </Author>
      – <Author>
          <FirstName>David</FirstName>
          <FamilyName>Powers</FamilyName>
        </Author>
      </Authors>
      <Publisher>friends of ED</Publisher>
    </Book>
  </Books>
```

Safari, on the other hand, strips out all the tags and just presents the text without any line breaks or formatting. Notice also that the ISBN is not displayed, because it's inside the opening <Book> tag. Safari treats it the same way as an (X)HTML attribute—you can see it only if you view the page's source code.

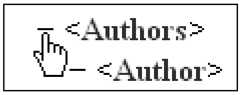


Foundation ActionScript for Flash 8 Kristian Besley Sham Bhangal David Powers friends of ED

Before we discuss the names of the tags, let's take a look at the hierarchical way the document is structured.

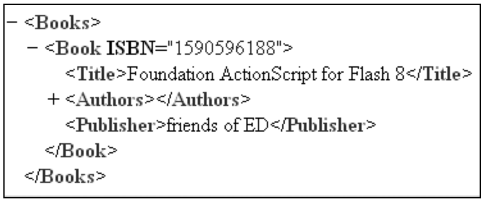
How an XML document is structured

Although it's not necessary to do so, most authors of XML documents indent lines in the way shown in book.xml, because it provides an important visual clue to the structure of the document. If you have loaded the page into Firefox or Internet Explorer, hover your mouse pointer over any of the minus symbols on the left side of the page. As you can see from the screenshot alongside, the cursor turns into a hand, indicating that it's hovering over some sort of link.



```
– <Authors>
  – <Author>
```

If you click one of the minus symbols, the browser will collapse the entire section of the document down to the equivalent closing link. The screenshot alongside shows what happens when you click the minus symbol next to <Authors>. The minus symbol turns into a plus sign, and the content between the opening and closing tags is hidden. To reveal the contents again, just click the plus sign.



```
– <Books>
  – <Book ISBN="1590596188">
      <Title>Foundation ActionScript for Flash 8</Title>
    + <Authors></Authors>
      <Publisher>friends of ED</Publisher>
    </Book>
  </Books>
```

So, what's the point of that? It doesn't make for exciting viewing, and it's not very user-friendly as a web page. Most XML documents aren't intended to be displayed on the Web in their raw state (unless they're written in XHTML, of course). They should be regarded as a logical way of storing data, rather like in a database. As you click each minus or plus sign, you are working with sections of the document, which are referred to in XML as nodes. A **node** can be any of the following:

- A section of other tags that lies between matching tags
- The text that lies between a single pair of matching tags
- An empty element
- The whitespace between tags

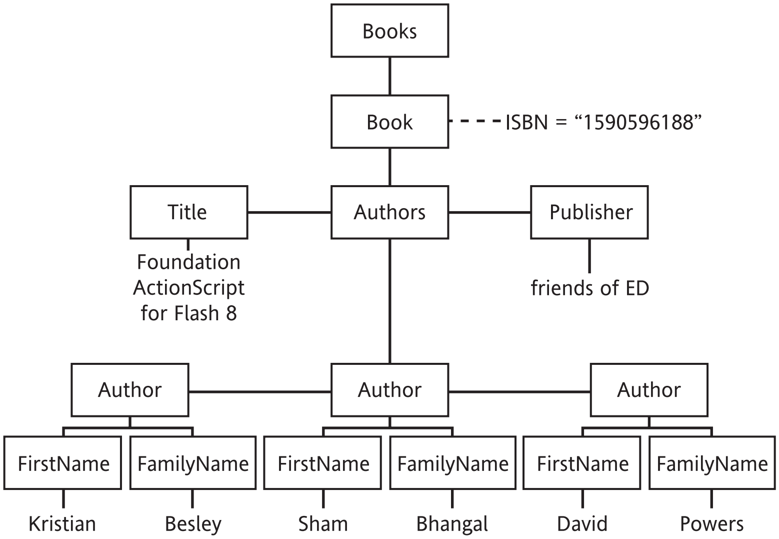
The file book.xml doesn't include any empty elements, but familiar examples from XHTML 1.0 are <br /> and <img />. All tags in XML must have a matching closing tag, and the forward slash before the closing angle bracket is simply a shorthand way of writing <br></br>. Understanding nodes is essential to working with XML in Flash and other web technologies, because you use them to navigate around a document.

Before you decide this discussion is getting far too technical, let's take a simpler look at the concept of nodes. The following illustration shows the same document redrawn in the more familiar shape of a family tree. The important thing to notice about this particular family is that it starts from a single parent (Books), which is known as the **root node**. As you burrow down into the family tree, a **parent node** can have many **child nodes**, so each child can have brothers and sisters (or **siblings**). Child nodes can also have many children of their own, but if you go back up the hierarchy, you'll notice that each child has only one parent. Finally, when you get to text between a pair of tags, that particular line of descent comes to an end. You have reached a **text node**, and text nodes cannot have children of their own.

That last sentence may have set you wondering how XHTML conforms to the rules of XML, because you constantly find text scattered between nonmatching tags like this:

```
<p>This will appear in <strong>bold</strong> font.</p>
```

The answer is simple: a text node can appear between any two tags. This explains why the whitespace between tags is regarded as a node. Some Flash developers mistakenly refer to this as the "whitespace bug," but as far as XML is concerned, any spaces or new lines between tags are text. Fortunately, Flash has a simple way of handling this, as you will see shortly.



XML borrows this genealogical terminology directly, so in book.xml, <Book> is a **child node** of <Books> and the **parent node** of <Title>, <Authors>, and <Publisher>. These last three are all at the same level of the family tree, so they are called siblings. There is even a pecking order among siblings: <Title> is known as the **first child** of <Book>, <Authors> is the **next sibling** of <Title>, and <Publisher> is the next sibling of <Authors>. <Publisher>, being the last of the children, can also be referred to as the **last child**.

Once you understand this structure and the associated terminology, handling XML in Flash becomes a lot easier, because the properties and methods of the ActionScript XMLNode class all derive their names from it.

Using the right version and encoding

The first line in book.xml looks like this:

```
<?xml version="1.0" ?>
```

This is the **XML declaration**, often also referred to as the **XML prolog**, which tells browsers and processors that it's an XML document. The XML declaration is recommended, but not required. However, if you do include it, the XML declaration *must* be the first thing in the document. Nothing else can precede it—not even whitespace.

Although version 1.0 isn't the most recent specification, it's the one you should use. Even the World Wide Web Consortium (W3C), which draws up the specifications for most web technologies, says so. XML 1.1 should be used only if you need its highly specialized features (for details, see www.w3.org/TR/2004/REC-xml11-20040204/#sec-xml11).

> *The XML declaration can also contain an* encoding *attribute. If you leave out this attribute, as we did in the previous example, XML parsers—including the one built into Flash—automatically use Unicode (*UTF-8 *or* UTF-16*).*

If your language is English and you never use anything else, you have nothing to worry about. Unaccented English uses the American Standard Code for Information Interchange (ASCII) format, which is a subset of UTF-8, so there is usually no need to specify an encoding. However, it's a different story if you need to use accented characters, such as in Spanish, French, and other European languages; if you're using mathematical symbols; or if you're building a Flash site that uses a completely different writing system, such as Chinese or Japanese.

Using non-English text with XML in Flash

If you're used to building websites in languages other than English, you'll be familiar with the <meta> tag used in (X)HTML to specify the correct encoding for a web page. The XHTML tag for Western European languages (including English) looks like this:

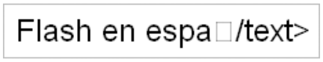
```
<meta http-equiv="Content-Type" content="text/html; ➥
      charset=iso-8859-1" />
```

Since the XML declaration accepts an encoding attribute, the natural assumption is that you should use the same character set as for your ordinary web pages, like this (the code is in iso8859_1.xml):

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<text>Flash en español</text>
```

There's no need to type the code yourself. All the following examples are in the download files, together with a SWF designed to load them at the end of this section.

When loaded into Flash, this produces the following mess onscreen:

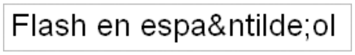


Flash en espa□/text>

This happens because region-specific encodings, such as `ISO-8859-1`, encode accented characters differently from the `UTF-8` that Flash expects. This leads many web designers to decide that the problem must be that they're not using HTML entities. So the next thing they try is this (see `html_entity.xml`):

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<text>Flash en espa&ntilde;ol</text>
```

This doesn't work, either, as the following screenshot shows:



Flash en espa&ntilde;ol

This happens because they're *HTML* entities, not XML entities. XML recognizes only the five HTML entities listed here:

- `&amp;` (&)
- `&apos;` (single straight quote)
- `&quot;` (double straight quote)
- `&lt;` (<, less than)
- `&gt:` (>, greater than)

Although XML doesn't recognize the remaining HTML entities, it does recognize their numeric equivalents. So you could use `&#241;` *instead of* `&ntilde;`*. However, it's not necessary, as you'll see in a moment.*

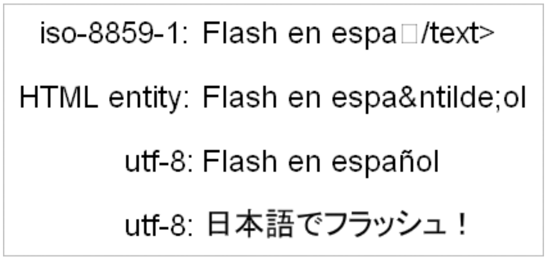
In desperation, they search the Internet for answers and come up with this "solution" frequently found in online forums—put the following code at the top of your Actions panel:

```
System.useCodepage = true;
```

This so-called solution should never *be used. Although it often appears to work when testing, it forces Flash Player to use the default language settings of the user's computer. So, if you're creating a Flash site in French, but a user from Japan visits your site, all the accented characters will be turned into Japanese kanji.*

The correct solution is disarmingly simple: **always save XML documents in UTF-8 encoding**. We'll do this in the next section.

As a demonstration, make sure that the following download files are all in the same folder: `html_entity.xml`, `iso8859_1.xml`, `utf8.xml`, `utf8_jpn.xml`, and `encoding_test.swf`. Double-click the SWF to launch it, and you should see the text in all four XML files displayed as shown in the following screenshot. On some systems, you may see a space instead of the little box in the first line. (The necessary characters have been embedded to enable you to view the Japanese sample file, but it won't work with other XML files in Japanese. Embedding the entire font increases the size of the SWF from 2KB to nearly 700KB!)

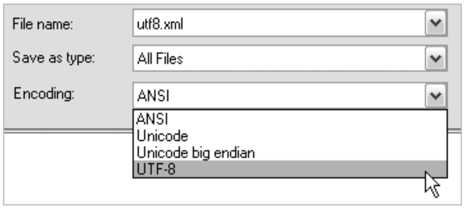


```
  iso-8859-1: Flash en espa□/text>
HTML entity: Flash en espa&ntilde;ol
       utf-8: Flash en español
       utf-8: 日本語でフラッシュ！
```

Saving XML files in UTF-8

Since XML files are written in plain text, saving XML files in UTF-8 is easy on both Windows and Mac OS X. You can use either a simple text editor, such as Notepad or TextEdit, or an XML editor.

Windows Notepad When you save a document, select All Files in the field labeled Save as type, and then choose UTF-8 from the Encoding options.

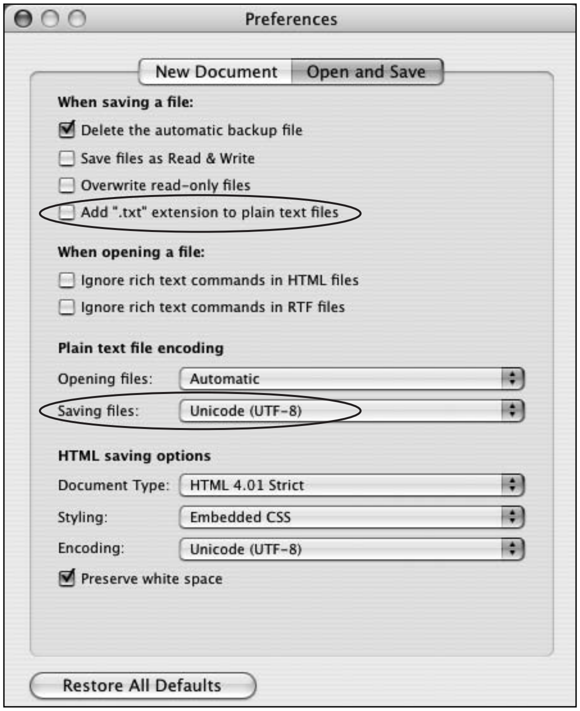


File name: utf8.xml

Save as type: All Files

Encoding: ANSI

TextEdit (Mac OS X) Open TextEdit ➤ Preferences ➤ Open and Save. Deselect the option labeled Add ".txt" extension to plain text files, as shown in the screenshot (this shows the Preferences panel in Tiger, which is laid out differently from Panther, but the options are the same). Then in the Plain text file encoding section, select Unicode (UTF-8) from the Saving files drop-down menu. By default, TextEdit saves files in Rich Text Format, which inserts unwanted formatting codes into your document. So, whenever you create a file, make sure you choose Format ➤ Make Plain Text.



When saving a file:

☑ Delete the automatic backup file
□ Save files as Read & Write
□ Overwrite read-only files
□ Add ".txt" extension to plain text files

When opening a file:

□ Ignore rich text commands in HTML files
□ Ignore rich text commands in RTF files

Plain text file encoding

Opening files: Automatic
Saving files: Unicode (UTF-8)

HTML saving options

Document Type: HTML 4.01 Strict
Styling: Embedded CSS
Encoding: Unicode (UTF-8)
☑ Preserve white space

Dreamweaver Dreamweaver MX, MX 2004, and Dreamweaver 8 have the option to create XML files (File ➤ New ➤ Basic Page ➤ XML). Dreamweaver automatically uses the same encoding as set in Preferences, but you can change this easily for any document by choosing Modify ➤ Page Properties and selecting UTF-8 from the Document encoding drop-down menu.

Dreamweaver is a good choice for creating XML documents, particularly if you already own it as part of the Studio MX 2004 or Studio 8 suite. One advantage of Dreamweaver is that it remembers the tags that you have created and offers automatic completion of ending tags. This not only saves time, but also cuts down on the likelihood of making mistakes.

Dedicated XML editors If you want to explore XML more seriously, you may want to consider a dedicated XML editor. Perhaps the best known is XMLSpy (`www.altova.com/products_ide.html`), which is available as a full-featured commercial version or in a free Home Edition (`www.altova.com/download_components.html`). XMLSpy runs on Windows only. Mac users may like to consider a multipurpose text editor called TextMate (`http://macromates.com`). It's inexpensive, and you can try it free for 30 days. You can find an exhaustive list of available alternatives, including several for Mac OS X, with brief reviews at `www.xmlsoftware.com/editors.html`.

> *Although the* `encoding` *attribute is not required if your XML document is saved as* `UTF-8`*, it's a good idea to include it in the XML declaration as a reminder of the encoding. Don't be fooled, though, into thinking that simply including* `encoding="utf-8"` *at the top of your page magically turns it into* `UTF-8`*. You must set the correct encoding in the software you use to edit XML, and* `UTF-8` *is the only encoding that displays XML correctly in Flash.*

How tags are used in XML

As you can see from `book.xml`, the tags give no indication as to how the document should look. This is because XML is intended primarily to store data in a hierarchical structure according to meaning, and without any reference to presentation. If you work in a large collaborative project, the tags will be decided centrally, or if you're creating a scientific document, you may use one of the specialist vocabularies, such as MathML. But most of the time, you make up your own tags. You can choose just about anything you want. As long as the tag names make sense to you, that's the main thing.

One of the main goals of XML is that it should be human-legible, so terseness is considered of minimal importance. Instead of using `<pub>`, which could mean publisher, publication date, or somewhere to get a drink, it's better to be specific. That's why `book.xml` uses `<Publisher>`. It means more typing, but there will be little doubt about what it means when you come back to look at the document in several months' time. XML is case sensitive, so the closing tag must be spelled exactly the same way—`</Publisher>`, not `</publisher>`.

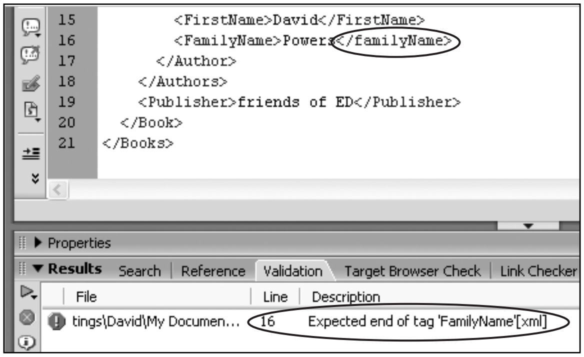
XML tags can not only be made up of alphanumeric characters, but also include accented characters and Greek, Cyrillic, Chinese, and Japanese characters—in fact, they can include any valid Unicode character. However, they cannot include any whitespace or punctuation other than the hyphen (-), underscore (_), and period (.). Nor can they begin with `xml` in any combination of uppercase or lowercase letters.

Checking that your document is well formed

The most important thing about an XML document is that it must be what is known as **well formed**. The main rules are as follows:

- There can be only one root element (root node).
- Every start tag must have a matching closing tag.
- Empty elements must have a forward slash before the closing angle bracket (/>).
- Elements must be properly nested.
- Attribute values must be in quotes.
- In a text node or attribute value, < must be replaced by `&lt;` and & by `&amp;`.

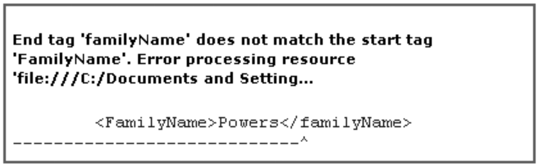
A big advantage of using a dedicated XML editor is that it will analyze your XML document and report any problems. To check an XML document in Dreamweaver, open the file and then choose File ➤ Check Page ➤ Validate as XML. The following screenshot shows the results of a check that discovered a misspelled closing tag in `book.xml` (`</familyName>` instead of `</FamilyName>`).



```
        <FirstName>David</FirstName>
        <FamilyName>Powers</familyName>
      </Author>
    </Authors>
    <Publisher>friends of ED</Publisher>
  </Book>
</Books>
```

File	Line	Description
tings\David\My Documen...	16	Expected end of tag 'FamilyName'[xml]

Even if you don't have an XML editor, it's very easy to check your documents in any modern browser. When the same page was loaded into Firefox, Internet Explorer, and Safari, all provided helpful error messages. The following screenshot shows the message displayed by Internet Explorer 6.



End tag 'familyName' does not match the start tag 'FamilyName'. Error processing resource 'file:///C:/Documents and Setting...

```
        <FamilyName>Powers</familyName>
----------------------------^
```

Right. That's plenty of theory to be going on with. Let's see how to load `book.xml` into Flash and access the data it contains.

> *A good starting place to learn more about XML is the XML FAQ, edited by Peter Flynn, at* `http://xml.silmaril.ie`*.*

Loading XML into Flash

Whenever you load an XML document into Flash, you need to do the following:

1. Create an XML object.
2. Tell Flash to ignore whitespace.
3. Load the document.
4. Check the document has loaded successfully and is usable.
5. Process—or **parse**—the data.

Let's use `book.xml` from the previous section (there's a copy in the download files for this chapter), load it into a Flash movie, and then extract some data.

Loading the book details into Flash

1. Open a new Flash document and save it as book.fla. Alternatively, you can look at the finished code in the download file of the same name. Make sure that book.xml is in the same folder.
2. Open the Actions panel and insert the following code:

```
// create XML object and load the XML data
var theBook:XML = new XML();
theBook.ignoreWhite = true;
theBook.load("book.xml");
theBook.onLoad = parseXMLData;
```

There's nothing complicated here. You begin by creating an XML object with the command new XML() and assigning it to a variable called theBook. In the same way as you used a sound object with the Sound class in the last chapter, you'll use this XML object to load and control your XML document by applying properties and methods of the XML and XMLNode classes.

The next line sets the value of the ignoreWhite property to true. This tells Flash to ignore all the whitespace between nodes in the XML document. It's as simple as that! However, you must always remember to include this line *before* you load the XML document. Otherwise it won't work.

The last two lines load book.xml and then assign a function called parseXMLData()—which you'll begin writing in the next step—as the onLoad event handler for your XML object. You can't do anything with XML data until it has been loaded, so the main processing code needs to go in the parseXMLData() function.

For this exercise, the XML document is in the same folder as the SWF, so you only need to pass the filename as the argument to the XML.load() method. If the file is in a different folder, use a relative or absolute path.

> *Increased security settings in Flash Player 7 and later normally prevent you from loading XML data from any other domain—in other words, both the SWF and the XML must be on the same website. The security restrictions are so tight that the basic URL must be exactly the same. For example, if your SWF is on* www.example.com*, the XML document cannot be on a subdomain, such as* games.example.com*, even if they are both on the same server and owned by you. To get around this restriction, you must create a cross-domain policy file or use a server-side solution such as PHP, ASP, ASP.NET, or ColdFusion.*
>
> *For details on how to create cross-domain policy files, open* Help ➤ Flash Help *and then select* Learning ActionScript 2.0 in Flash ➤ Understanding Security ➤ About domains, cross-domain security, and SWF files.

3. Now, let's start working with the XML. Add the following code *above* the existing script:

```
function parseXMLData(success:Boolean):Void {
  // make sure the data is loaded and usable
  if (success && this.status == 0) {
    trace(this);
  } else {
    trace("Problem loading book.xml");
    trace("The error code is " + this.status);
  }
}
```

The onLoad event of an XML object automatically returns a Boolean value indicating whether it succeeded in loading the data, so you pass this as an argument to the function assigned as the onLoad event handler—in this case, parseXMLData(). Testing this Boolean value in an if statement, however, only tells you whether the data was loaded. It doesn't tell you whether it was valid, so it's a good idea to use the status property of your XML object to check this. Since the XML object you created in step 2 is called theBook, you could use theBook.status. However, since the this keyword always refers to the object a function is applied to, it's a lot simpler to use this.status instead. A value of 0 means the XML document is OK.

If there's a problem loading the XML data, or if the data is unusable, the else clause will tell you what the problem is. Table 13-1 explains the meaning of the status codes:

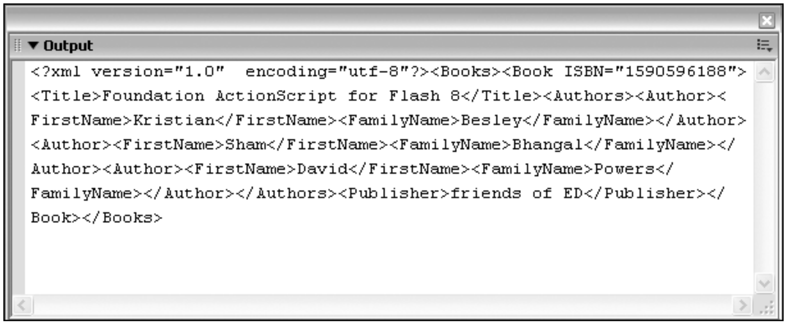
Table 13-1. Codes generated by the status property of an XML object

Error Code	Meaning
0	No error
–2	A CDATA section not properly terminated
–3	XML declaration not properly terminated
–4	DOCTYPE declaration not properly terminated
–5	A comment not properly terminated
–6	An XML element malformed
–7	Out of memory
–8	An attribute value not properly terminated (missing quote)
–9	A start tag not matched with an end tag
–10	An end tag without a matching start tag

If you tested that your document was well formed, you shouldn't see any of the error codes. The book.xml file doesn't use CDATA, a DOCTYPE declaration or comments, so you shouldn't see those error codes, either. XML comments, by the way, are exactly the same as those used in (X)HTML, and look like this:

```
<!--  There's no difference between XML and HTML comments. -->
```

4. Test your movie. As long as book.xml is in the same folder as your FLA, your Output panel should look something like this:



```
<?xml version="1.0"  encoding="utf-8"?><Books><Book ISBN="1590596188">
<Title>Foundation ActionScript for Flash 8</Title><Authors><Author><
FirstName>Kristian</FirstName><FamilyName>Besley</FamilyName></Author>
<Author><FirstName>Sham</FirstName><FamilyName>Bhangal</FamilyName></
Author><Author><FirstName>David</FirstName><FamilyName>Powers</
FamilyName></Author></Authors><Publisher>friends of ED</Publisher></
Book></Books>
```

Assuming you saw the contents of the XML document in the Output panel, congratulations. You're now ready to start extracting data. If you got an error message instead, go back and check that book.xml is well formed and that it's in the right location.

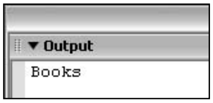
5. The way that you access data in an XML document is by using the family tree terminology that XML uses (described earlier in the section titled "How an XML document is structured"). The XML object, theBook, now contains the whole family tree, and the root node (<Books>) is referred to by using the firstChild property of the XMLNode class. To get the name of a node, you use the nodeName property. Since parseXMLData() is applied to theBook, you can still refer to it as this. Let's see this in action. Amend parseXMLData() like this:

```
function parseXMLData(success:Boolean):Void {
  // make sure data is loaded and usable
  if (success && this.status == 0) {
    trace(this.firstChild.nodeName);
  } else {
    trace("Problem loading book.xml");
    trace("The error code is " + this.status);
  }
}
```

Up to now, we've referred to the XMLNode *class only in passing. Now's the time to come clean and explain the difference between the* XML *and* XMLNode *classes. Technically speaking, the* XML *class is a subclass of* XMLNode. *It handles only the loading of XML data and sending it to external sources.* XMLNode *is the class that you deal with most of the time when extracting data from an* XML *object. Understanding this difference is important in deciding which data type to specify when defining variables.*

What's made life difficult is that, before the release of Flash 8, all documentation attributed most methods and properties to the XML *class. If you're new to ActionScript, this should make no difference, but you need to be aware that older books and online tutorials are likely to use the older—and therefore incorrect—classification.*

6. Test the movie again. You should now see the name of the root node in the Output panel, as shown in the screenshot alongside.

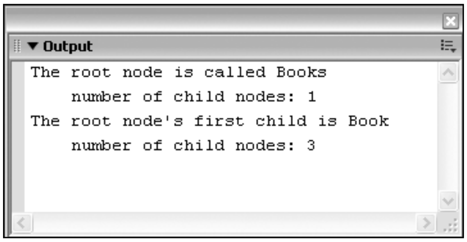


```
Books
```

7. Since the root node is the key to extracting all the data from an XML document, it's a good idea to assign it to a variable. You can then start querying its child nodes. A useful piece of information is the number of children a node has. Child nodes are always held in an array called—yes, you've guessed it—childNodes. So the length property of childNodes tells you how many nodes you need to process. Amend the code inside the if statement like this and test the movie again:

```
if (success && this.status == 0) {
  var theRoot:XMLNode = this.firstChild;
  trace("The root node is called " + theRoot.nodeName);
  trace("\tnumber of child nodes: " + theRoot.childNodes.length);
  trace("The root node's first child is " + ➥
theRoot.firstChild.nodeName);
  trace("\tnumber of child nodes: " + ➥
theRoot.firstChild.childNodes.length);
} else {
```

The Output panel should display the result shown in the following screenshot.

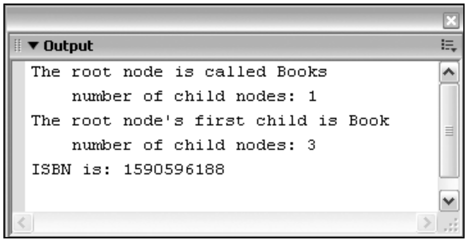


```
The root node is called Books
    number of child nodes: 1
The root node's first child is Book
    number of child nodes: 3
```

8. If you look at book.xml, you'll see that the <Book> tag has an attribute called ISBN. The way you extract the value of an attribute is with the attributes property followed by the name of the attribute. Add this to the end of the if statement in parseXMLData():

```
trace("ISBN is: " + theRoot.firstChild.attributes.ISBN);
```

The Output panel now shows you the book's ISBN number like this:



```
The root node is called Books
    number of child nodes: 1
The root node's first child is Book
    number of child nodes: 3
ISBN is: 1590596188
```

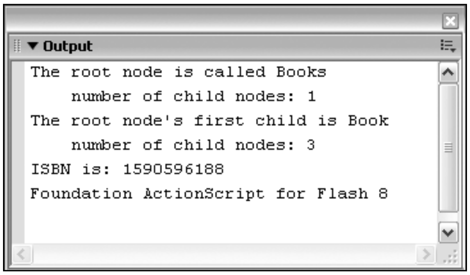
9. As you dig down into an XML document, you can find yourself dealing with lengthy, and potentially confusing, dot notation. For instance, to get to <Title>, you need the following:

```
theRoot.firstChild.firstChild
```

However, that gives you only the <Title> tag. The text node inside the tag is yet another firstChild, and to extract the actual text, you need to use the property nodeValue. This leaves you with this monstrosity:

```
theRoot.firstChild.firstChild.firstChild.nodeValue
```

If you add that in a trace() command to the existing if statement, the Output panel displays the title of the book like this:



```
The root node is called Books
    number of child nodes: 1
The root node's first child is Book
    number of child nodes: 3
ISBN is: 1590596188
Foundation ActionScript for Flash 8
```

As you have probably gathered, importing an XML document into Flash is simplicity itself. What's not so simple is extracting the data. You need to have a clear picture in your head of the data structure to be able to work through the different node levels. The more levels the XML document has, the more difficult it is to keep track of everything.

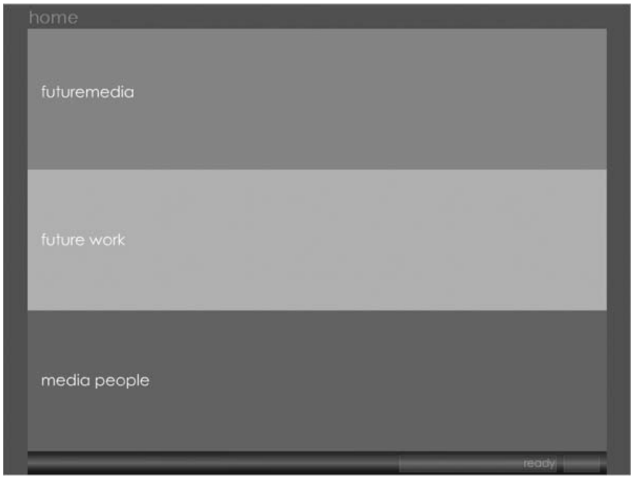
It's important to remember, though, that each set of child nodes is nothing more mysterious than an array. If you work through the arrays methodically, you can extract the data efficiently and then use it within your Flash application. A good technique is to work through each level and store the data in arrays or objects. In fact, you'll do exactly that when building the data structure for the Futuremedia site. Talking of which . . .

Book project: Controlling structure and content with XML

As you will know from the section titled "XML 101"—or if you're already up to speed on XML—there is no set structure to an XML document. As long as it's well formed, it can take whatever shape you like. The important things are that the structure is understandable and that it follows a predictable pattern. Consequently, there are several ways to represent the structure of the Futuremedia site as an XML document. The way we have chosen is by no means the only way you could do it, but it works. What's more, the ActionScript code that you'll use to handle it doesn't rely on you using the same tag names or the same number of elements in each node. It demonstrates just how flexible a dynamically driven site can be when controlled with XML.

Building the basic XML structure

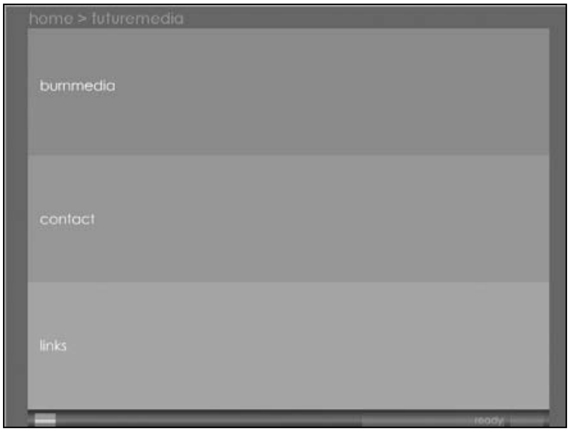
First, let's remind ourselves of what the opening page of the Futuremedia site looks like. It's something you've seen many times before, but you need to have it clearly in mind as you begin to build the XML document. The following image shows you how you can use the names of each strip on the home page as XML tags. The only difference is that spaces are not permitted in XML tags, so they have been replaced by underscores. The root node of the XML document has been called <home>. Using exactly the same names makes it easy to alter the navigational structure of the site. All will eventually become clear, we promise.



```
<?xml version="1.0" encoding="utf-8" ?>
<home>
   <futuremedia>



   </futuremedia>
   <future_work>



   </future_work>
   <media_people>




   </media_people>
</home>
```

Creating futuremedia.xml

1. Open a text editor, such as Notepad or TextEdit, or any program that has the capability of editing XML documents.
2. Enter the same code structure as shown in the previous illustration. The final code for this section is in `futuremedia.xml` in the download files for this chapter.
3. Save the file as `futuremedia.xml` in the same folder as your working document for the Futuremedia site. Make sure the file is saved in `UTF-8` format. (Refer back to the section titled "Saving XML files in UTF-8" if you're not sure how to do it.)
4. You now need to start building the content of each strip. The first one consists of three links, so you can follow the same basic structure as before. Insert the following code shown in bold between the `<futuremedia>` tags:



```
<futuremedia>
  <burnmedia>
  </burnmedia>
  <contact>
  </contact>
  <links>
  </links>
</futuremedia>
```

5. Each of these links leads to a content page. Because a content page leads nowhere else, it has no child nodes. (If you don't know what we mean by that, go back to the section titled "How an XML document is structured." This is vital to your understanding of the rest of the chapter, so now is the time to get the terminology clearly fixed in your head.)

   Since content pages have no child nodes, you could represent them in an XML document either as a text node or as an empty element. We've decided to take the second option, and we will use the empty element to store the page ID and URL as attributes.

   Amend the existing `<futuremedia>` section like this:

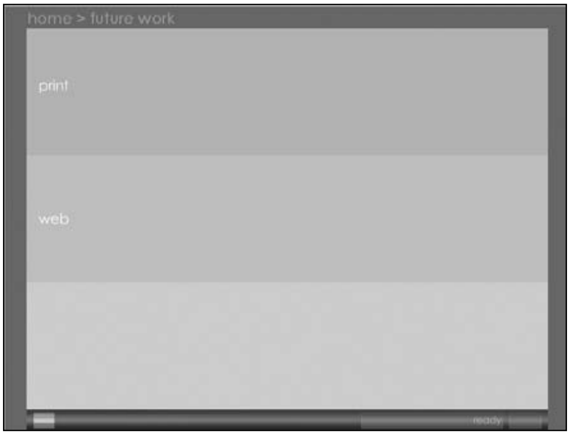
```
<futuremedia>
  <burnmedia>
    <contentPage id="futureTV" url=""/>
  </burnmedia>
  <contact>
    <contentPage id="map" url=""/>
  </contact>
  <links>
    <contentPage id="links" url=""/>
  </links>
</futuremedia>
```

At the moment, the url attributes don't have a value. Their role will be explained in the next chapter.

> *If you're familiar with XHTML, you may wonder why there is no space between the closing quote and the closing slash in any of the empty tags. The reason it's used in XHTML is for backward compatibility with older browsers. It's not necessary in an XML document, although there's nothing wrong with inserting a space either.*

6. The second top-level strip, <future_work>, leads to only two links at the next level; the third strip is empty. When building the XML document, there is no need to represent an empty strip. You only need to create nodes for elements that exist. This is where an XML structure really scores. Flash simply follows the XML. If something isn't there, it ignores it, as you'll see later once the two are integrated.

Amend the <future_work> node so that it looks like this:



```
<future_work>
  <print>
  </print>
  <web>
  </web>
</future_work>
```

7. The next level down in the second top-level strip is more complicated. The <print> node leads to two further links, <webdesign_books> and <other>, before reaching content pages. The <web> node, however, leads straight to a content page. The code for the <future_work> node needs to be amended like this:

```
<future_work>
  <print>
    <webdesign_books>
      <contentPage id="books" url=""/>
    </webdesign_books>
```

```
        <other>
          <contentPage id="draconis" url=""/>
        </other>
      </print>
      <web>
        <contentPage id="placeholder" url=""/>
      </web>
    </future_work>
```

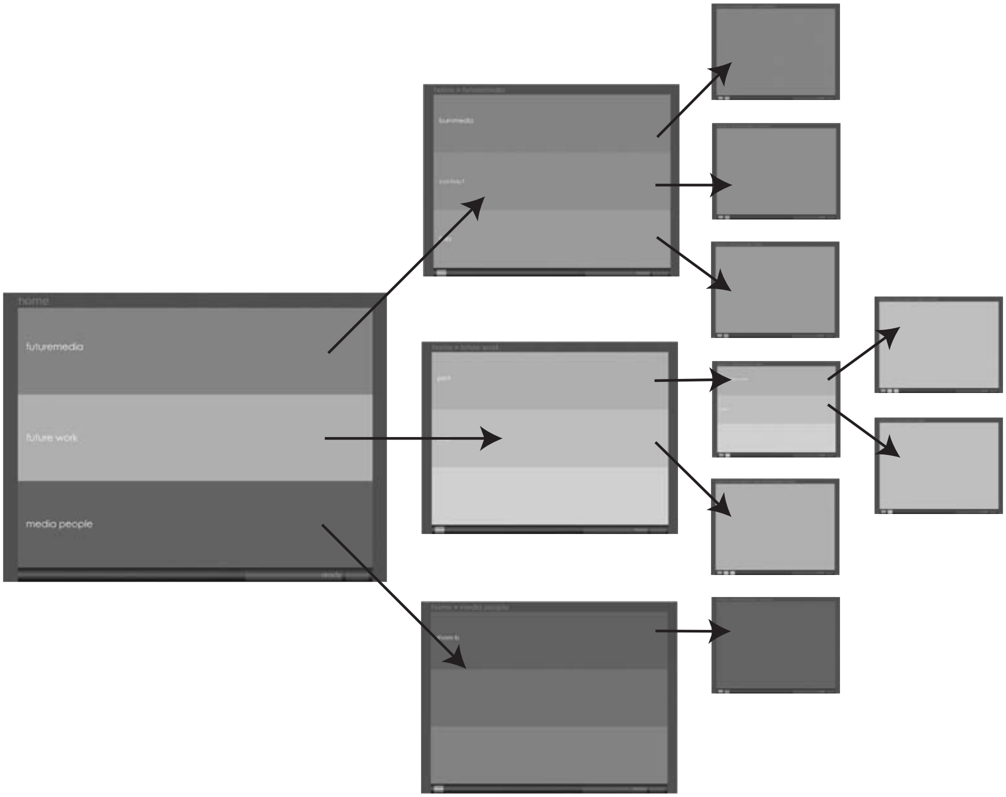
8. The third top-level strip, `<media_people>`, leads to just one link, `<sham_b>`, which in turn leads to a content page. Amend the `<media_people>` node like this:

```
<media_people>
  <sham_b>
    <contentPage id="shamb" url=""/>
  </sham_b>
</media_people>
```

9. Test that your XML document is well formed by using your XML editor to validate it or by loading it into a browser (see the previous section titled "Checking that your document is well formed"). The full listing for `futuremedia.xml` looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<home>
  <futuremedia>
    <burnmedia>
      <contentPage id="futureTV" url=""/>
    </burnmedia>
    <contact>
      <contentPage id="map" url=""/>
    </contact>
    <links>
      <contentPage id="links" url=""/>
    </links>
  </futuremedia>
  <future_work>
    <print>
      <webdesign_books>
        <contentPage id="books" url=""/>
      </webdesign_books>
      <other>
        <contentPage id="draconis" url=""/>
      </other>
    </print>
    <web>
      <contentPage id="placeholder" url=""/>
    </web>
  </future_work>
  <media_people>
    <sham_b>
      <contentPage id="shamb" url=""/>
    </sham_b>
  </media_people>
</home>
```

At this stage, it may still be difficult to understand how this maps to the structure of the site, so let's take another look at the structure of the strips. As the following diagram shows, you always progress through a page displaying three strips until you reach a content page. Sometimes not all strips will have a link—at the second level, the first navigation tree has three, the second has two, and the third has just one. Another complication is that level 3 doesn't consist only of content pages—one page links down even further to level 4. Believe it or not, this doesn't matter. As long as your XML document preserves the structure of no more than three link pages within any strip, you can vary the content as much as you like.



The reason for this becomes clearer if you spread out the XML code with lots of tab spaces like this:

```
           Level 1        Level 2                 Level 3                              Level 4
<home>
       <futuremedia>
                      <burnmedia>
                                  <contentPage id="futureTV" url=""/>
                      </burnmedia>
                      <contact>
                                  <contentPage id="map" url=""/>
                      </contact>
                      <links>
                                  <contentPage id="links" url=""/>
                      </links>
       </futuremedia>
       <future_work>
                      <print>
                                  <webdesign_books>
                                                                        <contentPage id="books" url=""/>
                                  </webdesign_books>
                                  <other>
                                                                        <contentPage id="draconis" url=""/
                                  </other>
                      </print>
                      <web>
                                  <contentPage id="placeholder" url=""/>
                      </web>
       </future_work>
       <media_people>
                      <sham_b>
                                  <contentPage id="shamb" url=""/>
                      </sham_b>
       </media_people>
</home>
```

What you have created, in effect, is just a series of nested arrays. By looping through the arrays, you can extract the data and process it accordingly. The first priority is to get the XML data into Flash.

Loading the XML data into Flash

Now that you have created the XML document, it's time to load it into your Flash movie.

Loading futuremedia.xml into your movie

Continue working on the FLA from Chapter 11. Alternatively, use the `futuremedia_code03.fla` file from the download files from that chapter. If you just want to follow the code, take a look at `futuremedia_code04.fla`, which contains the finished code for this chapter.

1. Open the Actions panel on frame 1 of the actions layer and add the following to the main code just before the stop() command at the bottom of the script:

```
// Set UI colors...
tricolor.setCol(color0, color1, color2);
// load XML data
var theData:XML = new XML();
theData.ignoreWhite = true;
theData.load("futuremedia.xml");
theData.onLoad = parseXMLData;
// now sit back and let the event scripts handle everything!
stop();
```

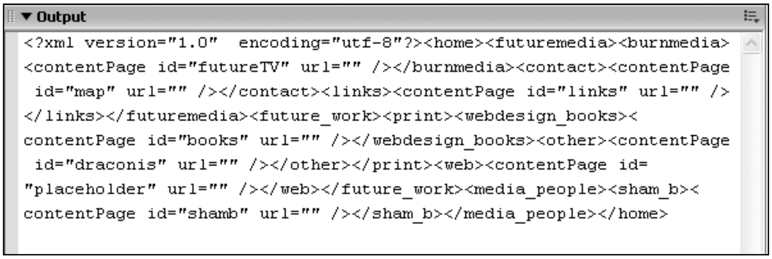
   There's nothing complicated here. It's almost exactly the same code you used in the earlier exercise "Loading the book details into Flash." The only differences are in the name of the XML object and the file being loaded.

2. You now need to start writing the functions that will handle the XML data. Scroll up to about line 85 and insert the code highlighted in bold immediately after the end of the stripPage() function:

```
  this.inheritCols();
}
// DEFINE XML FUNCTIONS
function parseXMLData(success:Boolean):Void {
  // make sure data is loaded and usable
  if (success && this.status == 0) {
    trace(this);
  } else {
    trace("Problem loading XML document");
    trace("The status code is: " + this.status);
  }
}
// Define screen extents for later use...
Stage.scaleMode = "exactFit";
```

   Again, this is exactly the same as the previous exercise. You test whether the XML data has been loaded correctly, and if it has, display it in the Output panel.

   As long as futuremedia.xml is in the same folder as your FLA, your Output panel should look something like this:

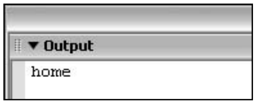


```
<?xml version="1.0"  encoding="utf-8"?><home><futuremedia><burnmedia>
<contentPage id="futureTV" url="" /></burnmedia><contact><contentPage
 id="map" url="" /></contact><links><contentPage id="links" url="" />
</links></futuremedia><future_work><print><webdesign_books><
contentPage id="books" url="" /></webdesign_books><other><contentPage
 id="draconis" url="" /></other></print><web><contentPage id=
"placeholder" url="" /></web></future_work><media_people><sham_b><
contentPage id="shamb" url="" /></sham_b></media_people></home>
```

3. The XML object, theData, now contains the whole family tree, and the root node (<home>) is referred to by using the firstChild property. To get the name of the root node, amend parseXMLData() like this:

```
function parseXMLData(success:Boolean):Void {
  // make sure data is loaded and usable
  if (success && this.status == 0) {
    trace(this.firstChild.nodeName);
  } else {
    trace("Problem loading XML document");
    trace("The status code is: " + this.status);
  }
}
```

4. Test the movie again. You should now see the name of the root node in the Output panel, as shown in the screenshot alongside.



```
home
```

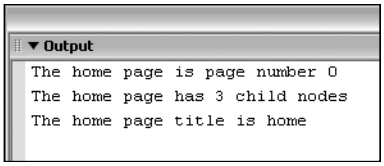
5. To keep track of all the information that you need to pick up on your journey through the family tree, it's a good idea to create an object for each node and store its related data as properties. These are the three properties and what they will contain:

   - pageNum: The page number that the node will occupy in the data structure
   - theNodes: An array of its child nodes (if any)
   - breadcrumb: The title used as a breadcrumb trail across the top of the page

   So let's do that for the home page. You'll also need arrays for the second, third, and fourth levels. Change parseXMLData() so it looks like this:

```
function parseXMLData(success:Boolean):Void {
  // make sure data is loaded and usable
  if (success && this.status == 0) {
    var homePage:Object = new Object();
    var secondLevel:Array = new Array();
    var thirdLevel:Array = new Array();
    var fourthLevel:Array = new Array();
    homePage.pageNum = 0;
    homePage.theNodes = this.firstChild.childNodes;
    homePage.breadcrumb = this.firstChild.nodeName;
    trace("The home page is page number " + homePage.pageNum);
    trace("The home page has " + homePage.theNodes.length + ➥
" child nodes");
    trace("The home page title is " + homePage.breadcrumb);
  }
}
```

Note that the else clause has been removed from the function. You can use trace() only for debugging, and the final code will populate the site with default data that will be displayed if the XML data fails to load correctly. In fact, that's the next stage we'll move on to. But first, test your movie. The Output panel should display the same information as shown in the screenshot alongside.



```
The home page is page number 0
The home page has 3 child nodes
The home page title is home
```

At the moment, it may seem as though you're making painfully slow progress. However, it's essential to keep verifying that you're getting the data you expect. Until you get the hang of extracting data from an XML hierarchy, it's easy to go charging off in the wrong direction. By checking each step as you go, it's easier to spot where you have gone wrong.

Creating the Futuremedia site's data structure

We'll leave XML for a while and build the default data structure that was outlined in Chapter 11. Let's briefly recap how it will work.

How the page array is structured

The site content is controlled by an array called page that contains details of what goes in each strip or content page. Each element in the page array is an object with five properties as follows:

- pType: This is the page type. It can have only two possible values, contentPage or linkPage, so it can be stored as a string.
- pLink: This is an array containing the page number that each strip links to. The problem is that not every strip links to another page, so you can have any number of links, from zero to a maximum of three. The way around this is to treat "no link" the same as an actual link, so *you always have the same number of links*. You can't use 0, because that's a valid page, so –1 will be used to represent the absence of a link.
- pTitle: This is an array of page titles. You need four: one for the navigation title (at the top left of the page) and one each for the three strips.
- pID: This is a string containing the page ID, and it is required only for a content page.
- pURL: This is a string that defines where a content page gets its content from.

Unused properties are set to null, which is computer-speak for "no value."

Populating the data structure with default data

1. Begin by defining your data array, page. This needs to go toward the bottom of the main script, just above the definition of the XML object. When the site first loads, a function called navigationData(), which uses page, needs to run. So you can also add a line for that at the same time, like this:

```
// Set UI colors...
tricolor.setCol(color0, color1, color2);
// Define and populate data structure
var page:Array = new Array();
navigationData();
// load XML data
var theData:XML = new XML();
```

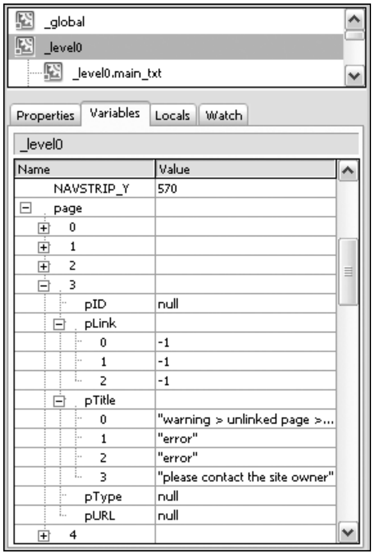
2. The place to add the navigationData() function is around line 85, between the end of the stripPage() function and the parseXMLData() function that you started building in the preceding section. Insert the following code:

```
  this.inheritCols();
}
// DEFINE SITE NAVIGATION DATA
function navigationData():Void {
  // define and populate page data structure
  for (var i:Number = 0; i < 50; i++) {
    page[i] = new Object();
    page[i].pType = new String();
    page[i].pLink = new Array();
    page[i].pTitle = new Array();
    page[i].pID = new String();
    page[i].pURL = new String();
    // populate with default values...
    page[i].pType = null;
    page[i].pLink = [-1, -1, -1];
    page[i].pTitle = ["warning > unlinked page > site error", ➥
"error", "error", "please contact the site owner"];
    page[i].pID = null;
    page[i].pURL = null;
  }
}
// DEFINE XML FUNCTIONS
function parseXMLData(success:Boolean):Void {
```

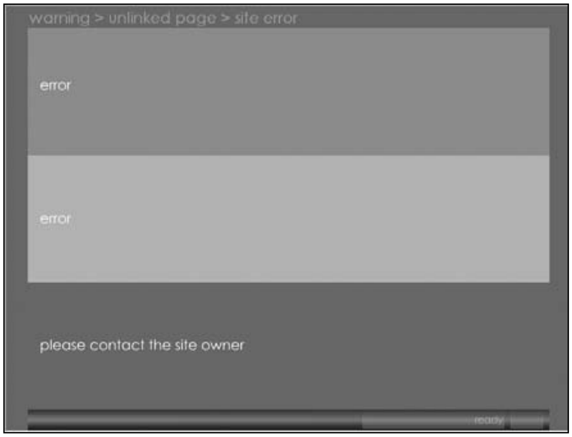
There's nothing new or difficult here. It's just a simple loop that creates a new object for each element in the page array, defines the five properties described earlier, and then assigns them default values.

3. Test the movie in debug mode. After clicking the continue button, look at the Variables tab for `_level0`. You'll see that each page object now has a set of default values, as shown alongside.

The defaults are for blank pages or broken links, so if the XML document fails to load correctly, visitors will see the following screen (it won't look like this yet).



```
_global
_level0
    _level0.main_txt

Properties | Variables | Locals | Watch

_level0
Name              Value
  NAVSTRIP_Y      570
⊟ page
  ⊞ 0
  ⊞ 1
  ⊞ 2
  ⊟ 3
      pID         null
    ⊟ pLink
        0         -1
        1         -1
        2         -1
    ⊟ pTitle
        0         "warning > unlinked page >...
        1         "error"
        2         "error"
        3         "please contact the site owner"
      pType       null
      pURL        null
  ⊞ 4
```



warning > unlinked page > site error

error

error

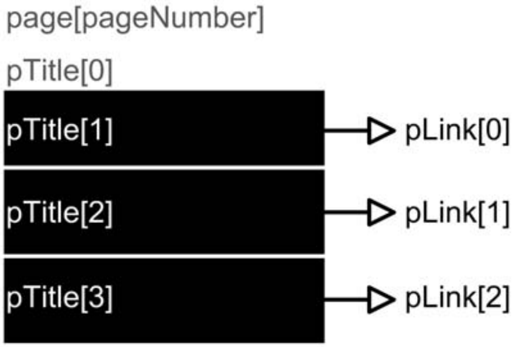
please contact the site owner

ready

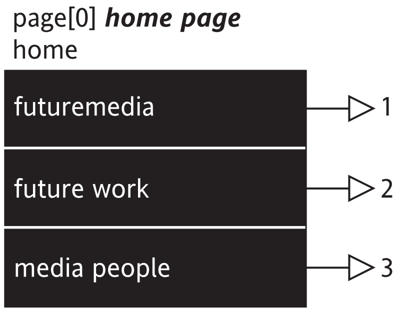
Populating the actual data values

You're finally ready to start working through the XML data to build the site. It should be noted that, because this is a book on ActionScript rather than Flash site design, we won't go too much into the reasoning behind the pages, page content, and the nature of the general site page structure, although it's fairly self-evident.

To begin, let's have a look at how the pages will look and how they'll link. The basic pattern is to have three links and four titles for each `page[pageNumber]`, like this:



So the home page looks like this in terms of data:



The home page has three main areas, as follows:

- futuremedia tells users what Futuremedia is all about.
- future work includes a client list and links to personal work.
- media people includes information about the Futuremedia designers.

You can tell that this will be a link page from the diagram, because it has, um, *links*. However, your ActionScript won't have the benefit of this diagram, so it needs a way to determine whether it's dealing with a link page or a content page, and assigning the result to pType. It also needs a way of allocating the correct values to the pTitle and pLink arrays. So let's start coding again.

Extracting the data values from futuremedia.xml

Continue working with your work-in-progress FLA or follow the code in futuremedia_code04.fla.

1. The page numbers in the site will be allocated in sequence, so you need a couple of timeline variables to keep track of which page you're currently working on (dataPage) and the next available number (nextAvailable). The number of strips in the site is fixed, so it's also a good idea to declare a constant variable to use in loops. Add the following code to the XML initialization at the bottom of the script (around line 150):

```
navigationData();
// load XML data
var NUM_STRIPS:Number = 3;
var dataPage:Number = 0;
var nextAvailable:Number = 1;
var theData:XML = new XML();
```

2. Throughout the book, we've hammered home the need to break up your tasks into manageable chunks, so let's create a separate function to process the data. Remember, the XML data is actually a series of nested arrays, and there is going to be a lot of repetitive looping as you go through each one. Remove the three trace() commands at the bottom of the parseXMLData() function, replace them with the code highlighted in bold, and then add the skeleton for the processData() function immediately below parseXMLData(), like this:

```
    homePage.breadcrumb = this.firstChild.nodeName;
    processData(homePage, secondLevel);
  }
}
function processData(thisData:Object, nextLevel:Array):Void {
  // process the current page object
}
```

As you can see, the new function processData() takes two arguments. The first is the current page object that you're working on—in this case, homePage. The second argument is the name of the array that will hold the details of the next level of data as you burrow down the XML hierarchy. Don't worry if you don't follow this yet. You'll see how it works shortly.

3. The first thing that you need to find out about the current page is whether it's a link page or a content page. Again, let's modularize the process. Add the first bit of code into the processData() function, like this:

```
function processData(thisData:Object, nextLevel:Array):Void {
  //process the current page object
  // store the number of the current page being processed
  dataPage = thisData.pageNum;
  if (isContentType(thisData.theNodes[0])) {
    // if it's a content page, process accordingly
    setupContentPage(thisData);
  } else {
    // otherwise treat as a link page
    page[dataPage].pType = "linkPage";
    page[dataPage].pTitle[0] = thisData.breadcrumb;
  }
}
```

This code starts by assigning the number of the current page object that's being processed to the timeline variable dataPage. By using a timeline variable, you can access the same value in other functions without the need to pass it as an argument.

Next, thisData.theNodes[0] is passed to a new function called isContentType(). theNodes is the name of the property you created earlier to store an array of a page object's child nodes. Since a content page never has any siblings, you need check only the first node in the array. This function isContentType() simply checks the name of a node and returns true if it's contentPage. Otherwise it returns false. You'll create the code for that and for setupContentPage() in the next step.

If the page isn't a content page, it must be a link page, so you can set the value of pType to linkPage. Finally, the first element in the pTitle array is set to the value stored in the current page object's breadcrumb property.

4. You can test this code as soon as you have defined isContentType() and its associated function, setupContentPage(). So add them now, immediately after the end of processData().

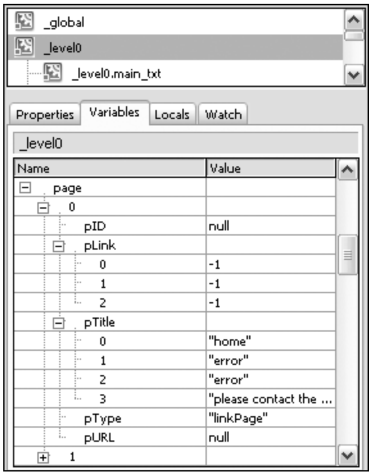
```
function isContentType(theNode:XMLNode):Boolean {
  if (theNode.nodeName != "contentPage") {
    return false;
  } else {
    return true;
  }
}
function setupContentPage(thePage:Object):Void {
  page[dataPage].pType = "contentPage";
  page[dataPage].pTitle = [thePage.breadcrumb, "", "", ""];
  page[dataPage].pID = thePage.theNodes[0].attributes.id;
  page[dataPage].pURL = thePage.theNodes[0].attributes.url;
}
```

These two functions grab all the information that you need for a content page.

The first function, isContentType(), takes a single argument, the node that's being tested, and uses the nodeName property to check whether the tag name is contentPage. It returns a result of true or false.

The second function, setupContentPage(), sets all the properties necessary for a content page. The first element of the pTitle array is set to the value of the breadcrumb property, while the other three are set to empty strings. The last two lines of setupContentPage() use the attributes property of the XMLNode class to extract the values of the id and url attributes, which you access simply by using dot notation. Since a content page doesn't link to any other page, there is no need to set any values for the pLink array. The default values of –1 automatically indicate there are no links to follow.

5. Run the movie in debug mode. After clicking the continue button, select _level0 and view the content of the Variables tab. Expand page[0]. You should now see that the values for pTitle[0] and pType have been set to home and linkPage respectively, as shown in the screenshot. Since this is a link page, pID and pURL will remain null. Now you need to set the three page numbers for pLink and the remaining three elements for pTitle.



Name	Value
page	
0	
pID	null
pLink	
0	-1
1	-1
2	-1
pTitle	
0	"home"
1	"error"
2	"error"
3	"please contact the ...
pType	"linkPage"
pURL	null
1	

6. Every link page has three strips, so you need to loop three times through its child nodes to extract the data for each one. If a page has fewer than three links, the value of nodeName for that strip will be null, and no data needs to be stored for it. Change the code in processData() like this:

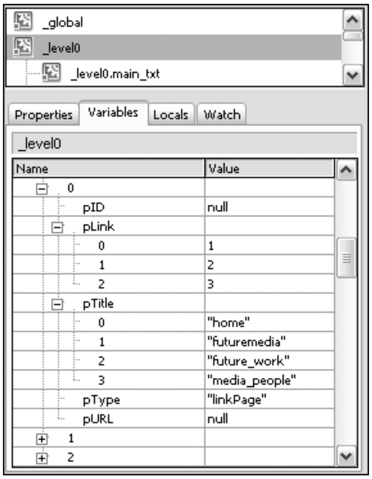
```
function processData(thisData:Object, ➥
nextLevel:Array):Void {
  // initialize variables
  var thisNode:XMLNode;
  //process the current page object
  // store the number of the current page being processed
  dataPage = thisData.pageNum;
  if (isContentType(thisData.theNodes[0])) {
    setupContentPage(thisData);
  } else {
    page[dataPage].pType = "linkPage";
    page[dataPage].pTitle[0] = thisData.breadcrumb;
    // now create the three strips
    for (var k:Number = 0; k < NUM_STRIPS; k++) {
      thisNode = thisData.theNodes[k];
      if (thisNode.nodeName != null) {
        page[dataPage].pTitle[k+1] = thisNode.nodeName;
        page[dataPage].pLink[k] = nextAvailable;
        nextAvailable++;
      } else {
        page[dataPage].pTitle[k+1] = "";
      }
    }
  }
}
```

The first addition is a local variable, thisNode, which keeps track of each of the three strips as you loop through them.

The loop begins by checking whether the value of nodeName is null. If it does have a name, it's added to the pTitle array and the next available page number is added to the pLink array.

On the other hand, if it doesn't have a name, the code skips the if statement and simply assigns an empty string to the current element of pTitle, leaving all other values at their default.

7. Run the movie in debug mode again. After clicking the continue button, select _level0 and the Variables tab. When you expand the values for page[0], you should now see the correct values for the first three strips displayed in pTitle, as well as links to pages 1, 2, and 3, as shown in the screenshot alongside.

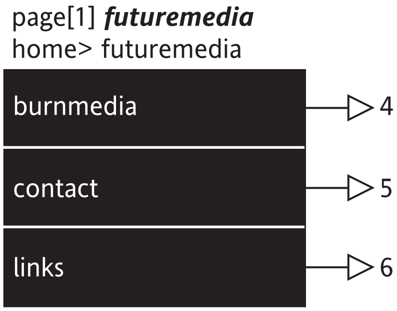


Name	Value
0	
pID	null
pLink	
0	1
1	2
2	3
pTitle	
0	"home"
1	"futuremedia"
2	"future_work"
3	"media_people"
pType	"linkPage"
pURL	null
1	
2	

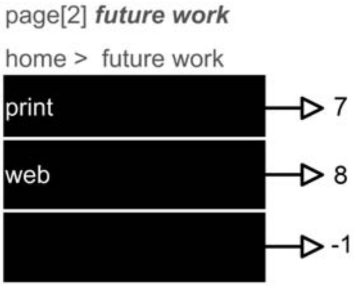
Moving to the next level

What you've managed to do so far has been fairly predictable. You've extracted details of the root node and its direct children. Moreover, you knew that there would be three strips to start with. From now on, things are much less predictable. Although each page has the capacity for three strips, there's no guarantee they'll all be occupied. However, the if statement in the loop you created in step 6 of the previous section should take care of that.

Next, you need to create an object to hold details of each strip and its own child nodes. Take, for example, page[1]. It's going to link to three link pages of its own, which will take the next three available page numbers: 4, 5, and 6.



The second strip in the home page (page[2]), on the other hand, links to just two link pages, like this:



By handling just one strip (or node) at a time, you break down the complexity. What's more, *you use exactly the same code to deal with every strip*. The data in your XML document is effectively a series of arrays. What you want, however, is to build pages, so you need to go through each array, assigning its data to a specific page. It doesn't matter if that page leads to further arrays—as long as you deal with each page in turn, you will keep everything in good order.

As you go through the loop in processData(), you need to store details of the current strip in an object that gives you access to the page number it occupies. Since the loop runs three times, you need to create three objects, each containing the following details of a page:

- Its number
- Its child nodes (if any)
- A page title built up from the page's parent and any other ancestors (its breadcrumb)

So, when you process the home page and store these details in the secondLevel array, secondLevel[0] is an object that contains all the details of page[1] and its descendents, secondLevel[1] holds the details of page[2] and its descendents, and secondLevel[2] holds the details of page[3] and its descendents.

Getting data from subsequent levels

1. Amend processData() like this:

```
function processData(thisData:Object, nextLevel:Array):Void {
  // initialize variables
  var thisNode:XMLNode;
  var tempObject:Object;
  dataPage = thisData.pageNum;
  if (isContentType(thisData.theNodes[0])) {
    setupContentPage(thisData);
  } else {
    page[dataPage].pType = "linkPage";
    page[dataPage].pTitle[0] = thisData.breadcrumb;
    // now create the three strips
    for (var k:Number = 0; k<NUM_STRIPS; k++) {
      thisNode = thisData.theNodes[k];
      if (thisNode.nodeName != null) {
        page[dataPage].pTitle[k+1] = ➥
removeUnderscore(thisNode.nodeName);
        page[dataPage].pLink[k] = nextAvailable;
        tempObject = new Object();
        tempObject.pageNum = nextAvailable;
        tempObject.theNodes = thisNode.childNodes;
        tempObject.breadcrumb = thisData.breadcrumb +" > " ➥
+ removeUnderscore(thisNode.nodeName);
        nextLevel.push(tempObject);
        nextAvailable++;
      } else {
        page[dataPage].pTitle[k+1] = "";
      }
    }
  }
}
```

This code adds a local variable called tempObject that is used to store the properties of the strip currently being processed. Inside the loop, it stores the value of nextAvailable as its page number. Then its child nodes are stored in theNodes, and its nodeName is added to the existing breadcrumb. There's also a new function, removeUnderscore(), which does exactly what its name suggests.

Finally, inside the loop all the values stored in tempObject are added to the array represented by the nextLevel variable, using the push() method of the Array class. This adds a new element to the end of an array and is a useful way of adding new array elements when you don't know how many there already are.

Surely we know how many elements there will be in this array, since it's going through a loop? Yes, we do this time, but that won't be the case when we get to the third level. This loop will be nested inside another one, so you can't rely on the loop counter.

The nextLevel variable is the second argument passed to processData(), so on this first occasion, it stores all the page objects in the secondLevel array. When the code is completed, it will later be used to store data in the thirdLevel and fourthLevel arrays.

2. The removeUnderscore() function is a simple string manipulation function that uses indexOf() to find the position of any underscore in a string. If it finds one, it replaces it with a space. Add the following code immediately after the setupContentPage() function (around line 158):

```
function removeUnderscore(theName:String):String {
  var thePos:Number = theName.indexOf("_");
  if (thePos != -1) {
    theName = theName.substring(0, thePos) + " " ➥
+ theName.substring(thePos+1);
  }
  return theName;
}
```

Note that this function will remove only one underscore. Unfortunately, ActionScript's string manipulation methods are quite basic. Although you could easily build a find and replace function, the idea here is to concentrate on working with the XML data.

3. The processData() function is now complete. You won't be able to see its full effect, though, until you pass each level of data to it in succession. Add the following code highlighted in bold to the last section of the parseXMLData() function (around line 115):

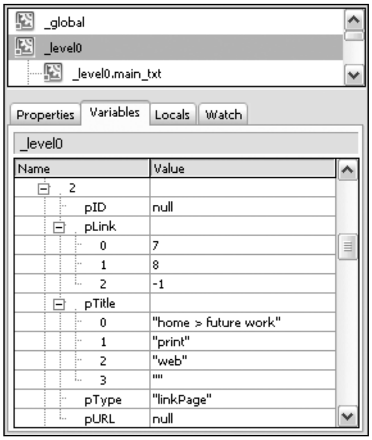
```
    processData(homePage, secondLevel);
    // process second level
    if (secondLevel.length > 0) {
      for (var i:Number = 0; i < NUM_STRIPS; i++) {
        processData(secondLevel[i], thirdLevel);
      }
    }
    if (thirdLevel.length > 0) {
      // process third level
      for (i = 0; i < thirdLevel.length; i++) {
        processData(thirdLevel[i], fourthLevel);
      }
    }
    if (fourthLevel.length > 0) {
      for (i = 0; i < fourthLevel.length; i++) {
        dataPage = fourthLevel[i].pageNum;
        if (isContentType(fourthLevel[i].theNodes[0])) {
          setupContentPage(fourthLevel[i]);
        }
      }
    }
  }
}
```

This series of if statements checks whether the next level array contains any data, and if it does, it runs a loop that iterates through the array, passing each page object in turn to processData() and assigning its results to the next level down. So, when you process secondLevel, the results are stored in thirdLevel, and thirdLevel results are stored in fourthLevel.

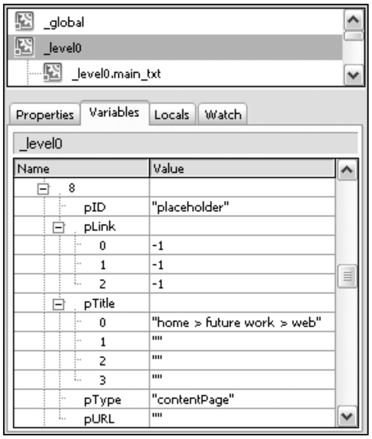
There are two points to note here. First, the loop that processes the second level is set to run the same number of times as you have strips in the tricolor movie clip, whereas the others use the length of the next level array. This is because the home page can never have more items than there are strips, but the subsequent levels have an unknown number of pages. The third and fourth levels have a theoretical maximum of 9 (3 × 3) and 27 (9 × 3), but futuremedia.xml as it currently stands generates only 6 and 2, respectively.

The other point is that the fourth-level loop processes only content pages. Forcing visitors to dig deeper than this for content is a sign of bad design. This is the three-click rule we talked about in Chapter 11. If visitors to your site can't get where they want in three clicks, you need to rethink your structure.

4. Test the movie in debug mode. After clicking the continue button, select _level0 and the Variables tab. You should now see that pages 0 through 11 have all been populated with data generated from futuremedia.xml. The following screenshots show the data stored for a link page and for a content page. The next stage is to get that information into the tricolor movie clip. If you want to check your code so far, compare it with futuremedia_code04.fla.



Name	Value
2	
pID	null
pLink	
0	7
1	8
2	-1
pTitle	
0	"home > future work"
1	"print"
2	"web"
3	""
pType	"linkPage"
pURL	null



Name	Value
8	
pID	"placeholder"
pLink	
0	-1
1	-1
2	-1
pTitle	
0	"home > future work > web"
1	""
2	""
3	""
pType	"contentPage"
pURL	""

Sanity check

At this stage, you're probably wondering how anyone can keep their sanity intact working with XML. We'll let you into a secret—it's something we've wondered at times, too. The difficulty lies in the fact that, as humans, we can look at an XML document and rapidly identify the section we want to go to. A computer can't do that; it needs instructions on how to get there.

When working with XML, you need to keep the family tree concept in mind at all times. It's also worth remembering that an XML parent is like a good mother or father—it always looks after its children. The way that you have just worked through futuremedia.xml is by dealing with one parent and its children at a time. You began with the root node <home> and queried it about its children, <futuremedia>, <future_work>, and <media_people>. Because you can deal with only one node at a time, you gave each of them a number and said, "Here, take this number and look after all your children until I get around to dealing with you."

After dealing with <home>, you then went back to <futuremedia> and discovered that it had at least one child of its own. Instead of inquiring about the children, you gave them a number of their own and put them in a queue to be dealt with later. To keep everything in order, it was more important to deal with the siblings of <futuremedia>. By doing so, you go down the family tree one generation at a time. If a particular branch of the family has "died out" in the next generation, you can safely omit it from your queries. If you attempt to go down each branch individually, you need to keep track of how far down you have gone so that you can find your way back. With a complex hierarchy, you would rapidly find yourself getting tied in knots.

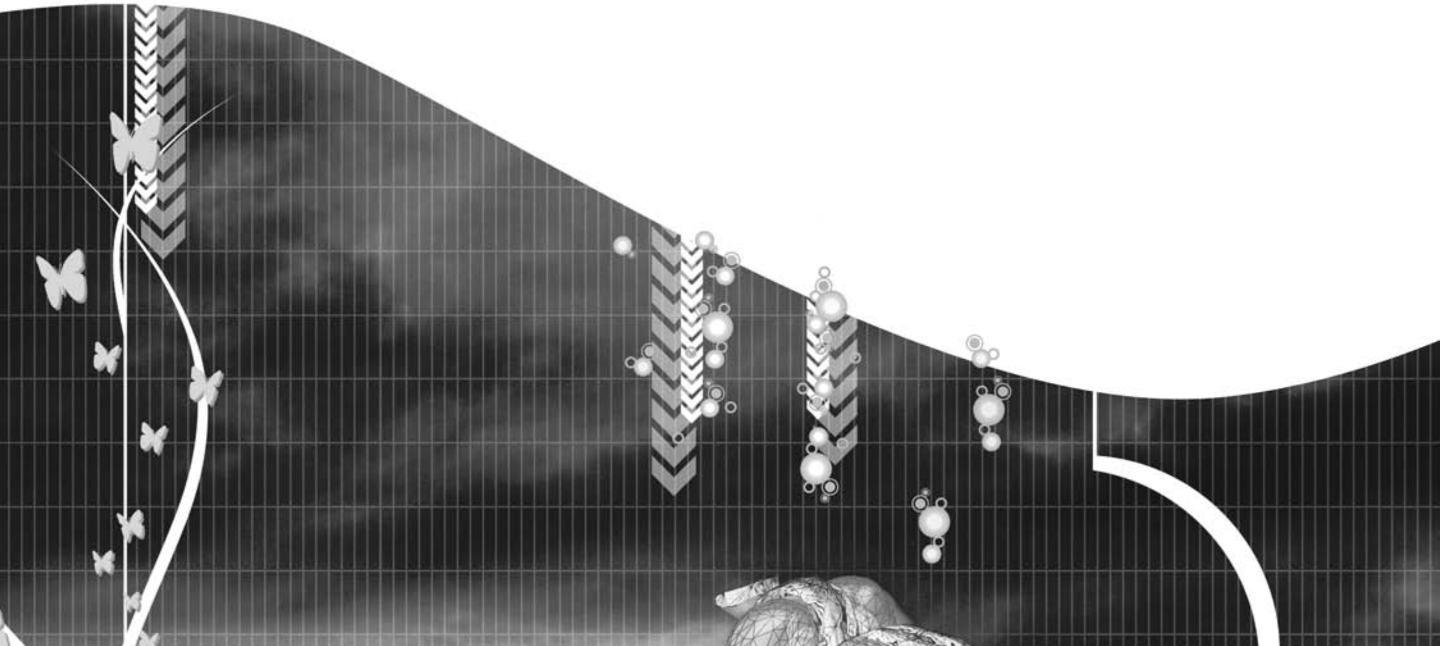
The crucial thing to realize is that the code you have written to process futuremedia.xml doesn't know anything about the tag names, nor does it know anything about the page numbers. In the previous edition of this book, all the links and page numbers were hard-coded into the ActionScript. Now, everything is generated automatically by the ActionScript looping through the data in the XML object. As long as you observe the rule of no more than three strips per link page and a maximum depth of four levels, the script will populate the page array with the correct data every time. So, although it takes time to learn how to work with XML data, it gives you great flexibility, which in the end saves even more time.

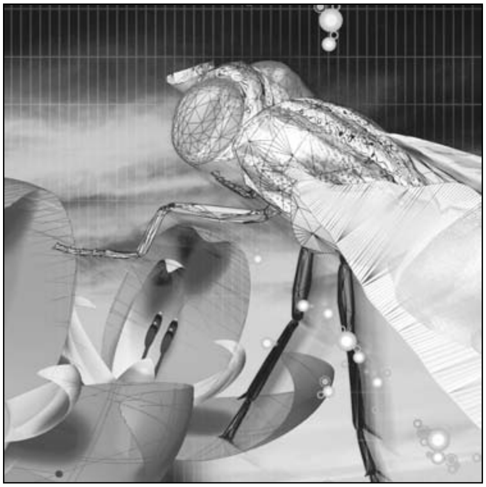
> *This chapter has only been an introduction to working with XML in Flash. There's a whole lot more to learn, including the creation of XML documents (although this needs to be done in combination with a server-side technology, such as PHP, ASP, ASP.NET, or ColdFusion). If we have whetted your appetite for more, take a look at* Foundation XML for Flash *by Sas Jacobs (friends of ED, ISBN: 1-59059-543-2).*

Summary

The XML and XMLNode classes are essential tools in creating dynamically driven websites with Flash. Although the methods and properties of these classes aren't difficult to use, learning how to navigate the family tree structure of an XML object often poses problems, particularly for nonprogrammers. What many people find off-putting is that you work with generic names, such as firstChild, rather than specific ones. But therein lies the strength and flexibility of XML—once you have designed the code to extract data from a particular type of XML document, the same code works for all of them.

The Futuremedia site is almost complete. You'll add the finishing touches in the next chapter by bringing the XML data into the user interface and tidying up other aspects of the site, including the addition of content. Then, in the final chapter, you'll roll up your sleeves to get down and dirty with version 2 Flash components as a prelude to dipping your toes in the wonderful world of OOP by building your own ActionScript classes.





Chapter 14

FINISHING THE FUTUREMEDIA CASE STUDY

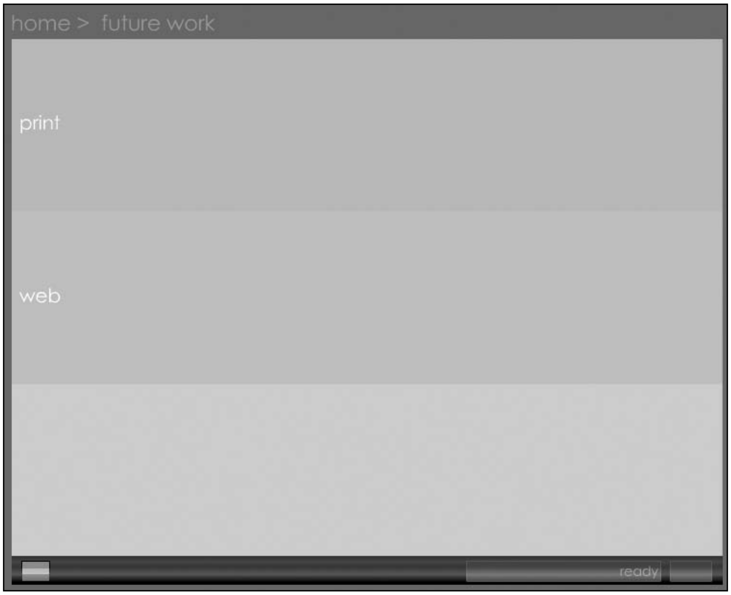
What we'll cover in this chapter:

- Getting the XML data into the user interface
- Loading placeholder content
- Tidying up the user interface
- Reviewing the structure of the code
- Adding a preloader
- Loading your own content into the site

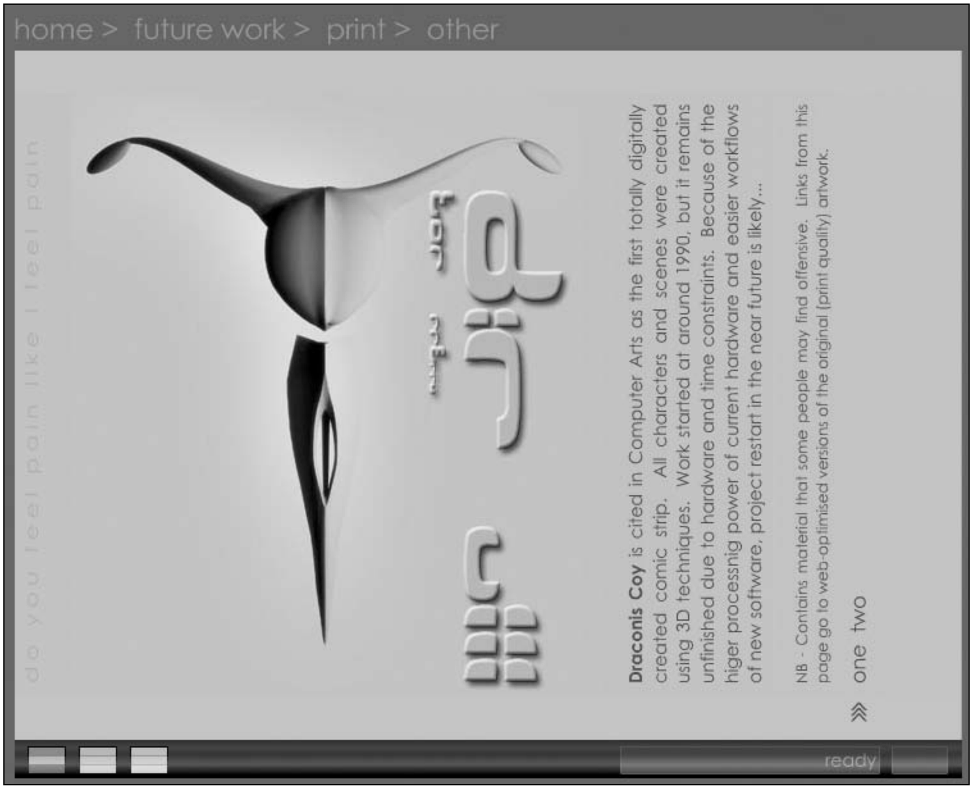
It's been a marathon journey since we first outlined the Futuremedia case study in Chapter 2. You've already coded more than 220 lines of ActionScript. You've got color and you've got motion, but still no content. Well, this is where everything finally comes together. The first task is to get the data imported from the XML document to drive the user interface. Then after testing the site with some placeholder content, you'll solve the problem of moving backward through the navigation system and add a preloader. We'll also step back to look at the processes that drive the site.

Getting the data into the user interface

OK, let's quickly recap. You've just processed the XML document and stored all the navigation data in a large multidimensional array called page. Your next task is to get Flash to use this information to populate your UI. If you look at the final site, you'll see that the page home > future work doesn't have a link for the third strip. When you navigate to this page and mouse over the three strips, only the first two show the hand cursor (signifying they're active links). The third doesn't act like a link because there is nowhere to go for this strip.



The page home > future work > print > other (shown next) shows content rather than links. Although when you first looked at the final site, you may have thought that such pages no longer contained the three strips, you now know that *all* pages contain the three strips. The three strips are disabled when they don't need to act as links. The function that does this disabling is the same one that disables the *one* strip in home > future work.


home > future work > print > other
do you feel pain like i feel pain
ready

Draconis Coy is cited in Computer Arts as the first totally digitally created comic strip. All characters and scenes were created using 3D techniques. Work started at around 1990, but it remains unfinished due to hardware and time constraints. Because of the higer processnig power of current hardware and easier workflows of new software, project restart in the near future is likely...

NB - Contains material that some people may find offensive. Links from this page go to web-optimised versions of the original (print quality) artwork.

one two

So, you need three new functions:

- `populateTitles()` to add all the text titles
- `populateEvents()` to make strips act like links when necessary
- `unpopulateEvents()` to make the strips stop acting like links

Populating the titles

Continue working with the same FLA. Alternatively, use `futuremedia_code04.fla` in the download files for Chapter 13. If you just want to follow the code, take a look at `futuremedia_code05.fla` in the download files for this chapter. Make sure that `futuremedia.xml` is also in the same folder as your FLA.

1. Add this code after the end of the function fadeText(), around line 60:

```
  tricolor.strip2_txt._alpha = fadeValue;
}
// BUILD PAGE AND DYNAMIC UI ELEMENTS
function populateTitles(pageIndex:Number):Void {
  main_txt.text = page[pageIndex].pTitle[0];
  tricolor.strip0_txt.text = page[pageIndex].pTitle[1];
  tricolor.strip1_txt.text = page[pageIndex].pTitle[2];
  tricolor.strip2_txt.text = page[pageIndex].pTitle[3];
}
// DEFINE MAIN SETUP SCRIPTS
function stripEvents():Void {
```

   This new function takes an argument, pageIndex, and uses it to retrieve the titles for page[pageIndex]. It then populates the four text fields with the appropriate text strings, as specified by the data generated from the XML document.

   Apart from the fact that you don't have a pageIndex argument anywhere yet, the text titling is sorted out. We'll come back to pageIndex a bit later. Next stop: the linking.

2. Add the following code below the populateTitles() function you just added:

```
function populateEvents(pageIndex:Number):Void {
  if (page[pageIndex].pLink[0] != -1) {
    tricolor.strip0_mc.onRelease = navigate;
  }
  if (page[pageIndex].pLink[1] != -1) {
    tricolor.strip1_mc.onRelease = navigate;
  }
  if (page[pageIndex].pLink[2] != -1) {
    tricolor.strip2_mc.onRelease = navigate;
  }
}
```

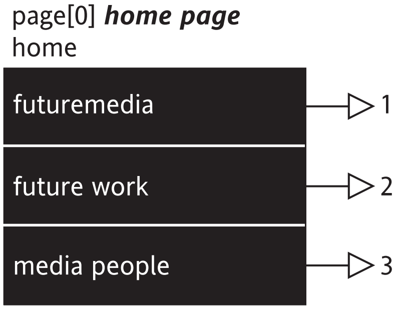
   All this function does is look at the current page (page[pageIndex]) and add an onRelease event that's defined as our old friend navigate(), so that if a link exists for a strip, it's made to act like a button (thus enabling the user to follow the link). It does this *unless* any strip has a –1 link defined for it.

3. Now add the third function directly below populateEvents(). This next block of code simply does the reverse of its counterpart—it *removes* all the events:

```
function unpopulateEvents():Void {
  delete tricolor.strip0_mc.onRelease;
  delete tricolor.strip1_mc.onRelease;
  delete tricolor.strip2_mc.onRelease;
}
```

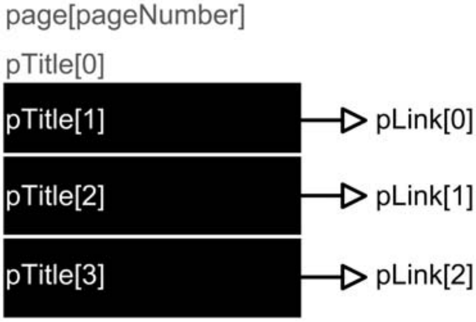
   This function deletes all the button events, so that the UI no longer responds to button clicks on the three strips. You now need to modify your existing functions to use the new events.

What you don't yet know is the page you're on. Well, actually, you do know what page you're on when the site first loads—you always start at the home page, and that is page 0. When you navigate to another page, you should be able to tell where you are from the link data. Take another look at the sketch for page 0. You see immediately the bit of data you need: *the link number*. This tells you the page number right away. For example, if you click the bottom strip (media people), you go to page 3, and that becomes the new page number. When you pass pageIndex as an argument to the populateTitles() and populateEvents() functions, you want pageIndex to hold the value 3. This causes page 3 to be built as the next page.



Looking at the next, more general diagram, it becomes obvious that the current page is page 0 when you start. As soon as you click a strip, the new page becomes pLink[strip], where strip is the strip number (0, 1, 2).

Given that you know you're on page 0 to start with, you can assign this value to a variable that tells you the current page number. Let's call it currentPage.



4. Add this to the INITIALIZE section (at the end of the listing):

```
theData.onLoad = parseXMLData;
// Define number of first page...
var currentPage:Number = 0;
// now sit back and let the event scripts handle everything!
stop();
```

5. You need to make the call to populateTitles() right at the end of the parseXMLData() function. Amend the final section of code like this:

```
    if (fourthLevel.length>0) {
      for (i=0; i<fourthLevel.length; i++) {
        dataPage = fourthLevel[i].pageNum;
        if (isContentType(fourthLevel[i].theNodes[0])) {
          setupContentPage(fourthLevel[i]);
        }
      }
    }
  }
  populateTitles(currentPage);
}
```

The large number of curly braces at the end of this function may make it difficult for you to see what's happening, so let's boil down the function to its basic parts. This is what parseXMLData() now does in essence:

```
function parseXMLData(success:Boolean):Void {
  // make sure data is loaded and usable
  if (success && this.status == 0) {
    // process the XML data to populate the page array
  }
  populateTitles(currentPage);
}
```

The code inside the if statement runs only if the XML data is usable, but the call to populateTitles() is *outside* the if statement, so it will always run. This ensures that even if the XML data can't be loaded, the default values still are. Admittedly, this displays only error messages, but at least it tells visitors there is a problem.

6. Test your FLA. You'll see that you now have titles correctly showing on the home page. The only trouble is that when you click a link, the titles don't change. Well, they shouldn't—you haven't added that part yet.

7. The next step is to ensure that you *always* see the correct titles. You need to do this by changing currentPage when you move to a new page, and by populating the page based on the new value. The place to do this is at the end of posTransition(), once the position transition has occurred and the current titles have faded out. Add the following two highlighted lines to posTransition() (around line 30):

```
    tricolor.setCol(selectedCol, selectedCol, selectedCol);
    currentPage = page[currentPage].pLink[selectedStrip];
    populateTitles(currentPage);
    fadeColor = 0;
    fadeAlpha = 0;
  }
}
```

8. Test the FLA, and you'll see your titles appear. Unfortunately, you can still navigate to pages you shouldn't be able to (page –1, or empty links), and to fix that you need to look at integrating populateEvents() with the main code.

Amending the way events are handled

You need to change the way the strips work as follows:

- As soon as the user clicks a strip, use unpopulateEvents() to remove all events from all strips, leaving you with a blank slate.
- When the new page appears, turn strips that have a link into active buttons.
- Leave alone strips that don't have links, so they remain inactive.

How do you know when a strip has been clicked? Easy! navigate() is the onRelease event handler for each strip. A strip has been clicked *as soon as* navigate() *runs*.

Fixing the events

1. Add the call to unpopulateEvents() at the start of the function navigate(), as shown:

   ```
   function navigate():Void {
     unpopulateEvents();
     // set up target for animation transition
   ```

   This removes the button events from all strips in the tricolor movie clip.

2. Once you're at the new page, you want to add the events back in, but only for those strips that are valid links. This is done by the populateEvents() function that you created in the last section. You need to put the new events back onto the strips when the navigation has completed, and that occurs at the end of colTransition().

   Add the following call to populateEvents() at the end of colTransition() (around line 56 in the script):

   ```
       fadeText(100);
       delete this.onEnterFrame;
       populateEvents(currentPage);
     }
   }
   ```

3. Test this file, and you'll see that you're *almost* there. Although the pages now work for link pages, as soon as you hit a content page, the site still changes to the three strips. On a content page, the UI should simply act as a dumb background until the content is loaded, so you don't want it to split into the three colored strips. You need to make the UI read the pType value and act accordingly.

> *You have to do this because your page always remains the same. You never really go to a new page—you simply keep reusing the tricolor.*

Reading content pages

The color transition is controlled by the function colTransition(). You want to run this function only if the new page happens to be a link page. If it isn't, you don't need to fade the three strips in again.

1. To make this change, add the following code at the end of posTransition():

   ```
       currentPage = page[currentPage].pLink[selectedStrip];
       populateTitles(currentPage);
       // if this is a link page, create another tricolor
       if (page[currentPage].pType == "linkPage") {
         fadeColor = 0;
         fadeAlpha = 0;
       } else {
         // otherwise place the content
   ```

```
      placeContent();
    }
  }
}
```

If the current page happens to be a link page, you continue as before. If it isn't, you go to a new function, placeContent(). Don't worry—this is a very short function, and then you'll be done for this section.

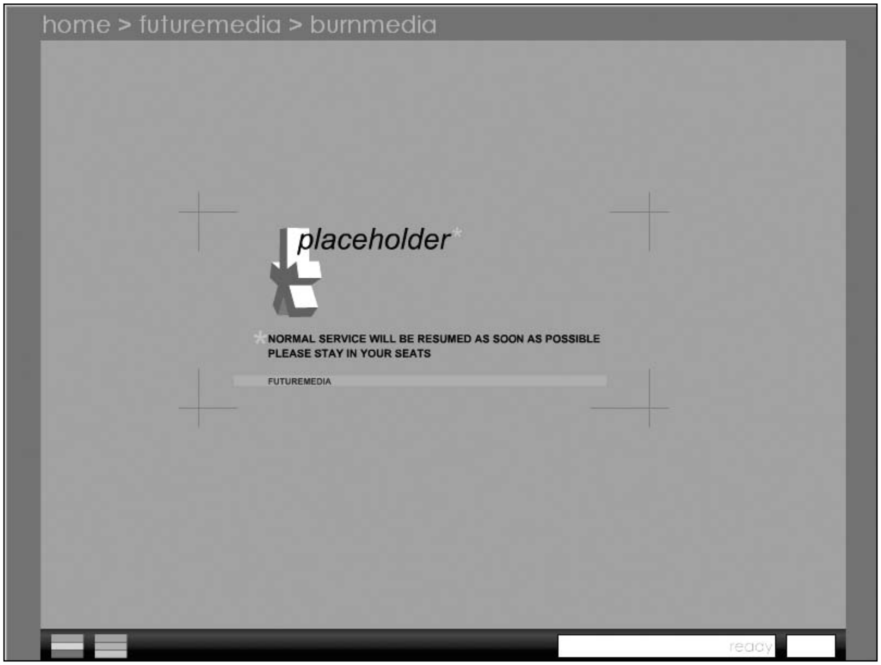
2. Add the following function below unpopulateEvents():

```
function placeContent():Void {
  _root.attachMovie("placeholder", "placeholder", 10000);
  _root.placeholder._x = BORDER;
  _root.placeholder._y = BORDER;
}
```

You'll find a copy of the placeholder movie clip in futuremedia_code05.fla. Import it into the Library of your work-in-progress FLA. If you decide to create a placeholder movie clip of your own, give it a linkage identifier of placeholder.

The reason we've assigned placeholder to a depth of 10000 is because the backward navigation, which you'll build shortly, needs to remove the content. It does this by creating an empty movie clip at the same depth, so it's essential to use a fixed number. By using such a high number, it's unlikely ever to clash with other assets you may decide to add at a later stage.

3. Test the movie again. This time, if you navigate to a content page, you should see the placeholder movie clip load into the page, as shown in the following screenshot.

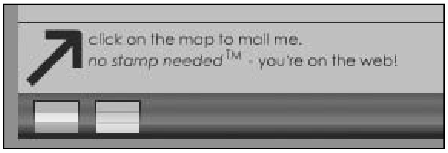

home > futuremedia > burnmedia

NORMAL SERVICE WILL BE RESUMED AS SOON AS POSSIBLE
PLEASE STAY IN YOUR SEATS

Wow, what a difference a few lines of code can make! Admittedly, all you have at the moment is a placeholder instead of content, but you now have an—almost—fully functional site. As long as your content pages are exactly the same size as the `tricolor` movie clip, all you need to do is to make a few changes to the `placeContent()` function, and they will load seamlessly into the site. A more immediate task is sorting out how to move backward through the navigation tree. At the moment, you can only go forward. You need a way back.

Adding the backward path

If you have a look at the final version of the site, you'll see that the back strip has a number of icons at the bottom left. Clicking one of these icons takes you back one or more pages through the current navigation, and clicking the first icon takes you back to the home page.


click on the map to mail me.
no stamp needed™ - you're on the web!

Back in Chapter 6, you created the symbol for the icons and stored it in the Library as `mc.icon` in the User interface ➤ active UI stuff folder. This symbol is almost exactly the same as `mc.page`, except there's no text and it has some lines to make the strips stand out better when scaled to a smaller size.

You need to create the functions to drive the icons, and then you need to integrate them with the existing code so that the functions are called at the appropriate times. So how do you do it? Well, there are a number of stages you need to consider.

- Initially, there are no icons. As soon as you click a strip, the icon to go back to the home page appears. As you move deeper into the site, more icons appear, each color-coded to look like a small version of the page it represents. Thus, a breadcrumb trail that shows you the path you've taken is created.
- As soon as you've used an icon to go backward, it's no longer needed and should disappear, as should the "back history" from that point onward. So clicking the home icon should result in all icons disappearing, because you've gone back all the way to the start.
- The final site has a separate animation to represent the user moving backward. Because this is different from the forward animation, you need a new function to handle it.

Making sure the UI knows where it needs to return

An obvious point, but perhaps the hardest one so far: when you click an icon, you need to re-create the page that it represents. You need three things to do this:

- The variable `currentPage` stores the number of the page currently being displayed, but it changes with each new page. Each icon needs to preserve the value of the page it is associated with, so it knows the correct page to return to.
- You need to know the colors of the new page you'll be going back to. Remember that the color of each page (except the home page) isn't predefined; it's calculated on the fly. However, the icon already "knows" the page colors, because it's colored using the same scheme.

- If the current page contains content, you need to delete the content from the screen before you go backward. An interesting question is "What if the new page you want to go back to also has content on it?" Well, that one caused us some worry initially, but the answer is actually very simple: *that never happens, so you don't have to worry about it*. The navigation is designed so that only the last page in any navigation tree is a content page; all the others leading up to it are link pages.

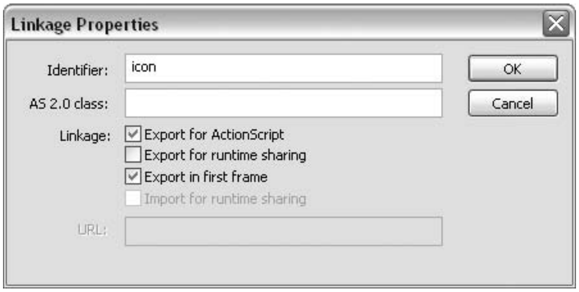
> *Although the back strip functionality looks and feels totally simple and intuitive, and is one of the little touches that makes the navigation look so sweet, you'll have realized by now that it's actually one of the hardest parts to design. Ergonomic simplicity is usually one of the more difficult things to code for all but the most basic UIs.*

Now that we've defined the problem, let's turn to the coding solution. You'll split the functionality into two parts:

- The part that looks after the building of the icon (and related data and event handlers), `buildIcon()`
- The part that looks after the animation transition, `iconTransition()`

Building the icons

1. The icons are placed on the back strip dynamically at runtime, so you need to give the `mc.icon` symbol in the Library a linkage identifier. You've done this plenty of times before, so the procedure should be quite familiar by now. In the Library panel, highlight mc.icon, right-click/Ctrl-click, and select Linkage from the context menu.
2. Select the Export for ActionScript option and change the value of Identifier to icon. Make sure that Export in first frame is checked, and click OK.



Linkage Properties

Identifier: icon

AS 2.0 class:

Linkage:
☑ Export for ActionScript
☐ Export for runtime sharing
☑ Export in first frame
☐ Import for runtime sharing

URL:

3. Since the icons will be used as a breadcrumb trail, you need to keep track of how many icons are displayed at any given time, so create a new variable, `iconCount`. Put it in the `INITIALIZE` block toward the end of the script (around line 250). You start off with none, so it's initialized to 0.

```
// Define navigation strip variables
var iconCount:Number = 0;
var NAVSTRIP_X:Number = BORDER;
var NAVSTRIP_Y:Number = BOTTOM;
```

4. It's a good idea to build the new icon as soon as a new page is created, and that means at the beginning of navigate() (right at the start of the listing):

```
// DEFINE EVENT SCRIPTS
function navigate():Void {
  unpopulateEvents();
  // build icon to get back to current page
  buildIcon(currentPage);
  // set up target for animation transition
```

5. The buildIcon() function is quite long, but it's similar to the code you've created for the tricolor movie clip and uses many of the same functions. We'll explain how it works in a moment. Put buildIcon() directly after unpopulateEvents() (around line 100). Here's the code:

```
function buildIcon():Void {
  // initialize
  var pageIndex:Number = 0;
  var iconDepth:Number = 100 + iconCount;
  var iconName:MovieClip = this.attachMovie("icon", "icon" + ➥
iconCount, iconDepth);
  // apply functions to icon
  stripCols.apply(iconName);
  // set colors, size, and position
  iconName._xscale = iconName._yscale=4;
  iconName.setCol(color0, color1, color2);
  iconName._x = NAVSTRIP_X+(40*iconCount)+10;
  iconName._y = NAVSTRIP_Y+5;
  // update iconCount by one for next icon
  // and store current page details onto the icon stack
  iconCount++;
  iconName.currentPage = currentPage;
  iconName.color0 = color0;
  iconName.color1 = color1;
  iconName.color2 = color2;
  iconName.iconCount = iconCount;
  // add the animation functions to icon
  // define icon button script
  iconName.onRelease = function() {
    // delete all icons after this one
    for (var i:Number = this.iconCount; i <= iconCount; i++) {
      _root["icon"+i].removeMovieClip();
    }
    // delete any content on the current page
    _root.createEmptyMovieClip("dummy", 10000);
```

```
            // zoom icon into the new current page
            scale = 4, x = this._x, y = this._y;
            xTarget = BORDER;
            yTarget = BORDER;
            scaleTarget = 100;
            delete this.onRelease;
            this.onEnterFrame = iconTransition;
          };
        }
```

How the buildIcon() function works

When you start, there are no icons on stage, and you have to place them on the stage with code. The first few lines of buildIcon() do this for you.

```
function buildIcon():Void {
  // initialize
  var pageIndex:Number = 0;
  var iconDepth:Number = 100 + iconCount;
  var iconName:MovieClip = this.attachMovie("icon", "icon"+ ➥
iconCount, iconDepth);
```

The instance name of each icon is icon plus the value of the new variable, iconCount. The first icon you create will thus be called icon0. Because you declared iconCount on the main timeline—and not inside the function—its value remains in memory. So, when the value of iconCount is incremented about ten lines further down, the next icon will be called icon1 and so on.

The depth is also based on iconCount. We've started the icon depths at 100 in case you need to attach other, more permanent stuff on _root.

> *Within this function,* this *is actually* _root. *Why? Well, the function isn't attached to anything, so it must be scoping the timeline it's defined on:* _root.

Once the icon has been created, you need to do quite a lot with it, so rather than refer to this["icon"+iconCount] every time, a reference to it is stored in a local variable, iconName. The first thing you need to do is change the strip colors within the icon. Fortunately, the icon and page movie clips are almost identical. They were deliberately designed that way so *you can use the same code to control the color changes in both*. The next bit of code attaches the stripCols function to the icon you just created.

```
stripCols.apply(iconName);
```

You then need to position and scale the icon. At the moment, the icon is the same size as the tricolor instance, so you need to make it much smaller. The following line does this by making it 4% of its original size:

```
iconName._xscale = iconName._yscale = 4;
```

Next, you need to color it:

```
    iconName.setCol(color0, color1, color2);
```

Then you need to position it on the back strip. Note that the position is dependent on the icon number, iconCount. The first icon is numbered 0, so it's located at NAVSTRIP_X (30), plus iconCount × 40 (0), plus 10—in other words, 40 pixels from the left edge. The next one will be located at 80 pixels. As you add more icons, the position changes so that icons don't end up overlapping:

```
iconName._x = NAVSTRIP_X+(40*iconCount)+10;
iconName._y = NAVSTRIP_Y+5;
```

Now here's the important part: you need to store the current page number so that you can get back to "here" if the icon is clicked later. You *could* be clever here and create an array a bit like page, which lists the backward path, but we prefer to keep it simple. The values that allow you to build the current page again are stored *inside the icon itself*. More correctly, the values that describe the current page are stored as *properties* of the icon instance.

```
iconName.currentPage = currentPage;
iconName.color0 = color0;
iconName.color1 = color1;
iconName.color2 = color2;
iconName.iconCount = iconCount;
```

Not only are the icons "visual breadcrumbs," but they're also "data breadcrumbs." The icons don't just look good, they are capable of storing data. One of the great advantages of the ActionScript MovieClip *class is that it's a dynamic class, which means that you can create new properties on the fly at runtime, and use them as variables that apply to a single movie clip instance. You don't have to create a separate object to store the properties as was needed with* page *and when looping through the XML data.*

The icons are also buttons, and that means you have to attach a button script to them. This is achieved by the next bit of code. The icon you just created is given an onRelease script.

```
iconName.onRelease = function() {
  // delete all icons after this one
  for (var i:Number = this.iconCount; i <= iconCount; i++) {
    _root["icon"+i].removeMovieClip();
  }
  // delete any content on the current page
  _root.createEmptyMovieClip("dummy", 10000);
  // zoom icon into the new current page
  scale = 4, x = this._x, y = this._y;
  xTarget = BORDER;
  yTarget = BORDER;
  scaleTarget = 100;
  delete this.onRelease;
  this.onEnterFrame = iconTransition;
};
```

What does this onRelease event handler do? Well, remembering what the icons do in the final site, when you click an icon, all icons to the right of it have to be deleted because they're no longer needed.

- Any content currently displayed has to be deleted.
- The icon has to grow to become the new page.

The first part is done here:

```
for (var i:Number = this.iconCount; i <= iconCount; i++) {
  this["icon"+i].removeMovieClip();
}
```

This deletes all icons between the value of iconCount stored inside the icon (in other words, the value that iconCount was when this icon was created) and the current value of iconCount on the main timeline. This has the effect of deleting all icons that were placed onscreen since this icon was created.

You then delete any content that may exist by creating an empty movie clip at depth 10000. Remember? This is the arbitrarily high depth used by the placeContent() function. Since two movie clips can't both occupy the same depth, the empty one kicks out anything that's already there, and since it's empty, it doesn't obscure anything at a lower depth.

```
_root.createEmptyMovieClip("dummy", 10000);
```

The last part of the script is much like the navigate() function. You set up for the position transition and delete the current onRelease event (so that the animation can't be stopped once it has started). You set up the target size and position you want the icon to end up as (at position (BORDER, BORDER) and size 100%, which is what it has to change to before it looks like the tricolor in position and size).

Implementing the icon transition

1. The buildIcon() function attaches iconTransition() as the onEnterFrame event handler for the icon. Add the following script immediately after buildIcon():

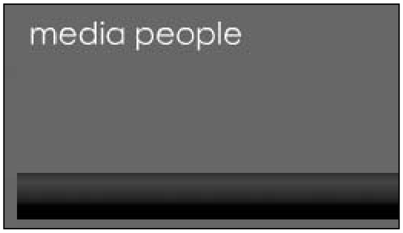
```
function iconTransition():Void {
  scale -= (scale-scaleTarget)/4;
  x -= (x-xTarget)/8;
  y -= (y-yTarget)/4;
  this._x=x, this._y=y;
  this._xscale = this._yscale = scale;
  // have we reached the target?
  if (Math.abs(y-yTarget)<3) {
    // change page to new page
    tricolor.setCol(this.color0, this.color1, this.color2);
    currentPage = this.currentPage;
    populateTitles(currentPage);
    populateEvents(currentPage);
    // update page color variables
    color0 = this.color0;
    color1 = this.color1;
    color2 = this.color2;
    // update the icon count and remove this icon
    iconCount = this.iconCount-1;
    this.removeMovieClip();
  }
}
```

This new function animates the icon as it grows to cover the current page. It's similar to the previous function, posTransition(), with one subtle sting in the tail: when the icon totally covers the old page, it *doesn't* become the new page. Instead, the icon is deleted and the current tricolor changes so that it looks like the icon. It might look like the icon grows and becomes the new page, but this isn't the case.

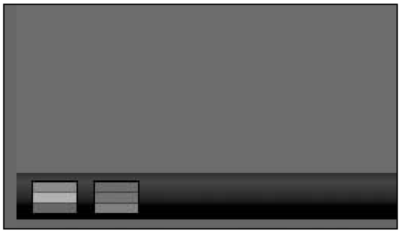
Why do you do this? Well, the site UI is written to animate tricolor. Changing this clip for a new icon clip causes no end of problems, so you do the sneaky switch-around to keep life simple.

The last thing iconTransition() does is set iconCount to the value of iconCount held in the icon (this.iconCount) minus 1. This resets the iconCount value on the main timeline to reflect the fact that some icons have been deleted.

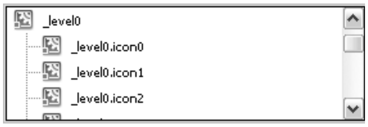
2. Test the movie or use futuremedia_code05.fla, making sure that it's in the same location as futuremedia.xml. You can now navigate backward and forward.
3. You can see the iconCount value and the way the icons store the current state of the navigation by testing the FLA in debug mode. When the site starts from the home page, you'll see that there are no icons.



As you navigate deeper into the site, you'll see the breadcrumb icon trail build up.

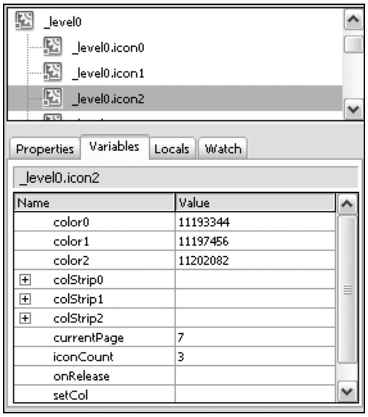


You can see the icon movie clips listed in the debugger, as shown in this screenshot:



```
_level0
    _level0.icon0
    _level0.icon1
    _level0.icon2
```

4. Look inside an icon in the debugger by selecting it and then opening the Variables tab. You'll see the local variables stored inside each icon instance (color0, color1, color2, currentPage, and iconCount). These variables are used to return to the page represented by the icon when the icon is clicked.



```
_level0
    _level0.icon0
    _level0.icon1
    _level0.icon2
```

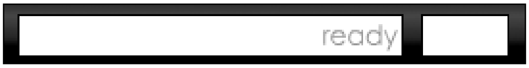
_level0.icon2

Name	Value
color0	11193344
color1	11197456
color2	11202082
⊞ colStrip0	
⊞ colStrip1	
⊞ colStrip2	
currentPage	7
iconCount	3
onRelease	
setCol	

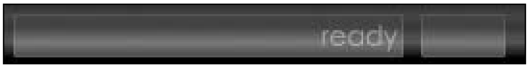
5. Finally, take a look at the way the variable `_level0.iconCount` changes as you interact with the icons.

Tidying up the user interface

There are a few glaring differences between the interface you have at the moment and the final one. For example, notice the appearance of the back strip. In the site at the moment, you have a couple of ugly text fields to the right of the back strip.



The version of the back strip in the final site looks significantly better.



Also, the text fields do nothing at the moment, so you need to start driving them with code.

The first task is to deal with the unsightly white backgrounds. You want to change two things:

- The text field background color
- The text field border color

Your first instinct is probably to use the Property inspector. Unfortunately, you can't. Try it and see, if you don't believe us. You need to use the ActionScript TextField class to do so. All the more reason for learning ActionScript: there are some properties that you can't access without it. Although we'll give you all the instructions, you should start getting in the habit of consulting the ActionScript 2.0 Language Reference to learn about the methods and properties of the available classes. You can access the ActionScript 2.0 Language Reference from Help ➤ Flash Help . Alternatively, you can download it in PDF format from www.macromedia.com/support/documentation/en/flash/. Don't be put off by the fact that it's nearly 1,400 pages long. It's full of example scripts, and if you use it like a dictionary, you'll be amazed at how rapidly your skills advance.

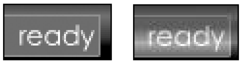
Improving the look of the dynamic text fields

1. Add the following code highlighted in bold, just after the point where you define and populate the data structure (around line 316):

```
// Define and populate data structure
var page:Array = new Array();
navigationData();
//Set up status text colors
status_txt.backgroundColor = 0x394444;
status_txt.borderColor = 0x5A5555;
statusValue_txt.backgroundColor = 0x394444;
statusValue_txt.borderColor = 0x5A5555;
// load XML data
var NUM_STRIPS:Number = 3;
```

This sets the background and border colors for the two text fields on the back strip to dark shades of gray, in keeping with the overall design (subdued color everywhere except the two focuses of attention: the central tricolor and the back icons).

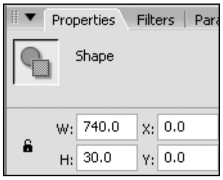
You still don't have exactly the same thing as the final site, though. The final back strip (below right) is shinier, with a subtle metallic effect.



This effect is created via a semitransparent movie clip that is placed over the back strip. You can see the symbol in futuremedia_code05.fla. Look in the Library and you'll find it in the User interface ➤ non active UI stuff folder as the movie clip strip mc.stripShine, which also has a linkage identifier of stripShine.

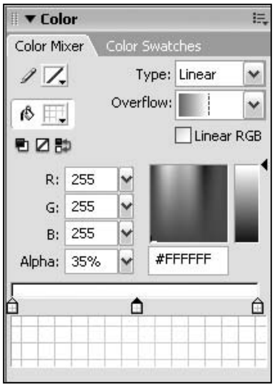
2. You need a copy of this movie clip. You can either take the easy route (drag and drop the clip from the futuremedia_code05.fla Library to your work-in-progress FLA) or create a version of your own. To create your own version, follow steps 3 through 8.

3. Create a new movie clip and call it `mc.stripShine`. Inside it, draw out a black, strokeless rectangle (the dimensions aren't important). In the Property inspector, give the rectangle the settings shown in the screenshot alongside. These numbers make the rectangle the same dimensions as the back strip.

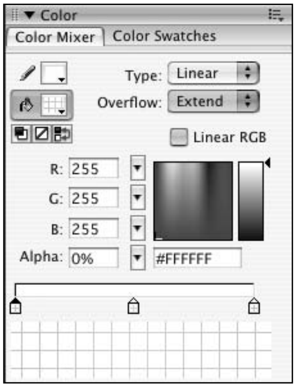


W: 740.0 X: 0.0
H: 30.0 Y: 0.0

4. Next, you need to give your black strip a semitransparent shine. Deselect the rectangle and open the Color Mixer panel (Window ➤ Color Mixer). Create a linear fill that is white with 35% alpha at the center, and white with 0% alpha at both edges.



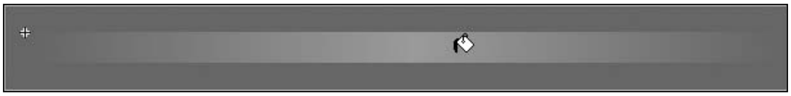
Type: Linear
Overflow:
□ Linear RGB
R: 255
G: 255
B: 255
Alpha: 35%
#FFFFFF



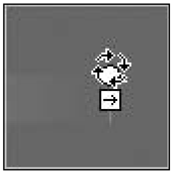
Type: Linear
Overflow: Extend
□ Linear RGB
R: 255
G: 255
B: 255
Alpha: 0%
#FFFFFF

We've included screenshots of both the Windows and Mac versions of the Color Mixer panel, because this is one of the rare occasions when they're different. If you look at the Mac version (on the right), you'll see that the Overflow drop-down menu is set to Extend, whereas the Windows version (on the left) has a gradient that fades toward a dotted line. The Overflow setting is new to Flash 8 and controls the colors applied past the limits of the gradient. There are three settings: Extend (default), Reflect, and Repeat. The Mac version uses just the names, whereas the Windows version tries to give you a visual representation of the effect. We'll leave you to experiment with this new toy later, but for this movie clip, you want the default (Extend).

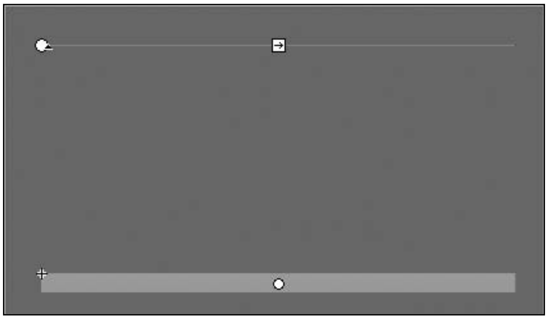
5. Apply this gradient fill to the rectangle using the Paint Bucket tool. The rectangle will take on a pale white glow at the center, fading to gray at both ends.



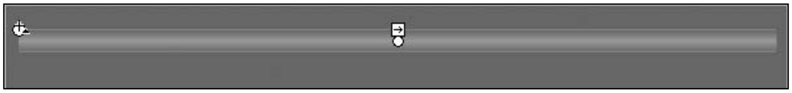
6. You need to rotate this fill so that it runs from top to bottom rather than from left to right. Select the Gradient Transform tool and click inside the rectangle. Hover your mouse pointer over the circular handle at the right end of the rectangle. When the mouse pointer turns into a circle of arrows, as shown in the screenshot alongside, hold down the mouse button and drag the mouse upward and to the left.



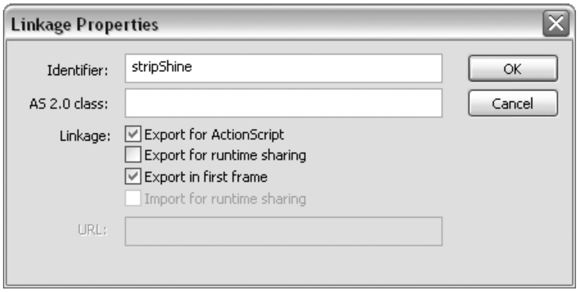
   Drag the mouse until the mouse pointer is directly above the center of the rectangle. When you release the mouse button, you should see a blue guideline parallel to the rectangle, as shown in the screenshot. If you have difficulty because the mouse pointer goes offscreen, change the zoom setting on the stage temporarily to 50%.



7. Now drag the handle in the middle of the blue guideline (it has a right-facing arrow on it) down toward the center of the rectangle so that the gradient is just a narrow strip across the full width, as shown in the next screenshot.



8. When you're done, give your `mc.stripShine` movie clip a linkage identifier of `stripShine`.



Linkage Properties

Identifier: stripShine

AS 2.0 class:

Linkage: ☑ Export for ActionScript

☐ Export for runtime sharing

☑ Export in first frame

☐ Import for runtime sharing

URL:

9. You can now use this strip to produce the metallic shine. Add the following lines after the code you added in step 1:

```
statusValue_txt.borderColor = 0x5A5555;
// position strip shine
this.attachMovie("stripShine", "shine", 200);
shine._x = NAVSTRIP_X;
shine._y = NAVSTRIP_Y;
// Load XML data
```

   Notice that you don't need to assign the attachMovie() to a movie clip reference, as you did with the icons. This is because you don't need to access the stripShine instance once it has been created—it's a static part of the UI.

> *Why go to all the trouble of attaching the* stripShine *movie clip to the stage dynamically? The reason is so that the shine appears above the icons, which are also created dynamically. Unless you place the shine strip dynamically, there's no way to make the shine appear above the icons. You can still see the icons because the* Alpha *setting of the gradient ranges from 0 to 35%.*

There are just a few more things that you need to add to the user interface. The amount of coding involved is very small, but it's probably worthwhile to stand back for a moment and look at the way the site works. Doing so will help you understand why the next code goes where it does.

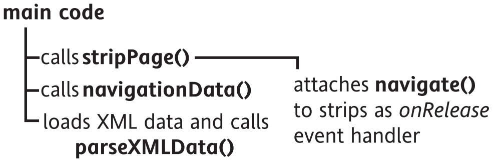
A great big sanity check

We imagine there are more than a few people out there who have been concentrating on each little thing they've done so far but can't step back and see the big picture. If you've just looked up and, after a few moments of reflection, said "Um . . . darn!" or choice words to that effect, then this sanity check has your name written all over it.

Because the code is very large, it can be difficult to see what's what anymore, so let's do a little pruning and unearth the original skinny pup that this big animal of a listing started off as. There are *only seven main functions* that do everything. Yes, they subcontract a lot of work to other functions, but the real work is done by three functions for the initialization and four functions for the actual navigation. Let's break this down visually before looking at the code again.

The initialization phase goes like this:

1. The main code calls stripPage(). This initializes the tricolor movie clip by setting its color and attaching navigate() to each of the strips as the onRelease event handler.
2. Next, navigationData() populates the data structure with default values.
3. The XML object, theData, loads the actual data from futuremedia.xml and passes it to parseXMLData() as soon as the data has finished loading. This builds the actual site structure.
4. Program flow then stops.


main code
calls stripPage()
calls navigationData()
loads XML data and calls parseXMLData()
attaches navigate() to strips as onRelease event handler

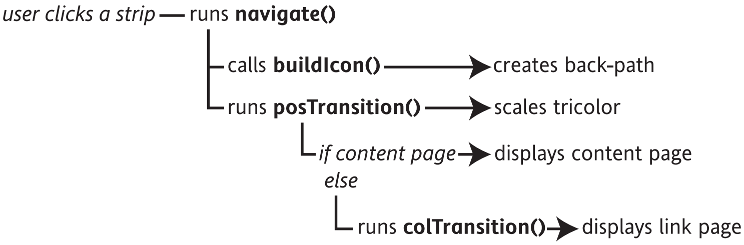
So how does stuff happen if no code is running anymore? The three strips within `tricolor` are now *buttons*. As soon as the user clicks one of the strips, the icons also appear, and these are also buttons. Program execution is kick-started by the user clicking any of these buttons. Clicking a strip takes the user forward in the navigation; clicking an icon takes the user backward in the navigation:

user clicks a strip
user clicks an icon

If the user clicks a strip, the program flow is as shown in the following image. The `onRelease` event handler, `navigate()`, is invoked, and this calls `buildIcon()`. It then makes `posTransition()` run as an `onEnterFrame` event handler to provide the first part of the navigation animation. This deals with the position and scale of the `tricolor` movie clip. It also triggers the loading of content if you've gone to a content page.

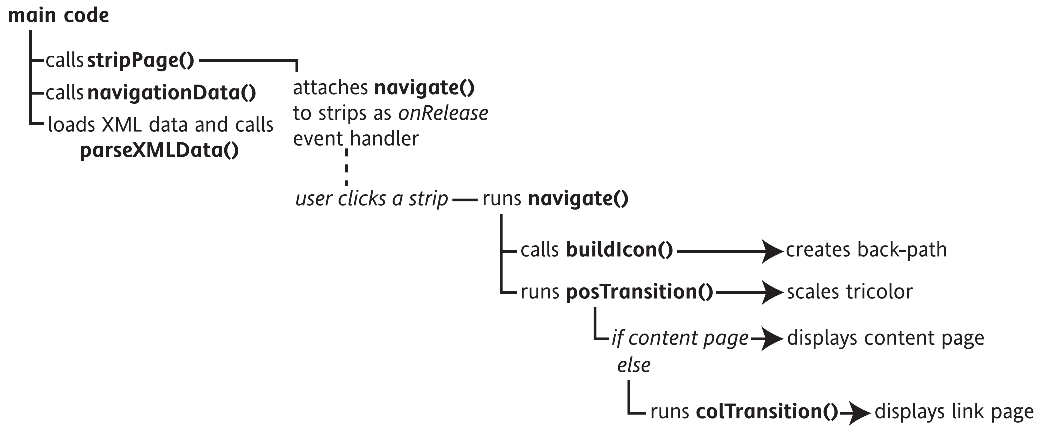
Once `posTransition()` has finished, a decision is made on whether `colTransition()` needs to run. This handles the color transition that fades in the three strips again, so it does need to run if the new page is a link page.

At the end of this flurry of execution, you have an icon created to go backward and either a content page or another link page.



If you're a little unsure about what's happening here, it's worth following this diagram while you test the UI so far.

You can now put together a full picture of what happens in the code.



Note that the preceding diagram shows what happens in code execution order, top left to bottom right. It also shows the time order from left to right. Finally, it shows the basic structure of the entire site design in terms of major building blocks (or, to use the terminology we've used previously, it shows you the relationships between all your black boxes). Quite a useful diagram to refer to!

You should now be able to understand the listing better by looking at the preceding diagram and picking out each function in the code listing. You can probably even recognize that skinny puppy again. For those who can't, here's a quick helper: it's *very* useful to read this listing while looking at the preceding image to see how it all fits together.

```
// DEFINE EVENT SCRIPTS
function navigate():Void {
  I am the onRelease handler for the three strips.
  I set everything off when you click a strip.
}
function posTransition():Void {
  navigate() makes me the onEnterFrame handler for the tricolor.
  I perform the first part of the navigation animation,
  positioning and scaling the tricolor.
  I also set off the loading of content if required.
}
function colTransition():Void {
  Once posTransition() has finished, it makes me the onEnterFrame
  handler for the tricolor if I am needed (i.e., if the current page
  is a link page). I then fade the strips back in.
}
```

```
function buildIcon():Void {
  I create icons to allow the user to move backward through the
  navigation, and I am called at the start of every forward
  navigation by navigate().
}
// DEFINE MAIN SETUP SCRIPTS
function stripPage():Void {
  I initialize the tricolor.
  I attach navigate() to each strip so that it becomes a button.
  This sets the whole site in motion.
}
// DEFINE SITE NAVIGATION DATA
function navigationData():Void {
  I set up the basic data framework for the user interface.
}
// DEFINE XML FUNCTIONS
function parseXMLData():Void {
  I process the XML data and use it to build the site structure.
}
// Initialize UI...
I am the main script.  I call stripPage() and then navigationData().
Then I sit back and let the event scripts handle everything!
```

Although the site may have started to look a little sluggish when you test it in Test Movie, try viewing the site in the browser and you'll see a different story. It's much quicker because Flash doesn't have to draw anything that ends up offscreen.

The Futuremedia design has been relentless in its approach—everything is modular. This is paying real dividends now at the end. Sticking almost pedantically to structured code has meant that you've been able to totally bypass one of the biggest issues with complex motion graphics: performance. The code modules work in a structured and sequential manner, and that means the site is fast.

If you don't follow this approach, you'll most likely ace the early stages of development, moving along at two or three times the speed. Only right here at the end game would you start to have problems. You would know your site is working, but you might be unsure of what bit of code is doing what, and whether it's running at times it shouldn't.

The Futuremedia site is designed in terms of the seven major code functions, and all of them have distinct interfaces. Among other things, the point at which each function starts running and stops is as regimented as an army on the march. When you say "Stop," everything really does!

We hope that has quelled any growing panic. Let's move on and start using this information.

Adding the status text messages

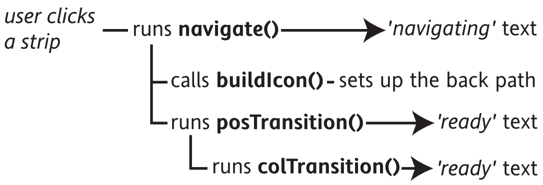
Although you've set the color of the two text fields at the bottom right of the back strip, they still don't do anything. In particular, the first one just sits there telling you it's "ready." Time to fix that. You'll change the message so that

- When the UI is animating, the status message will be navigating.
- When the UI isn't doing anything, the status message will be ready.

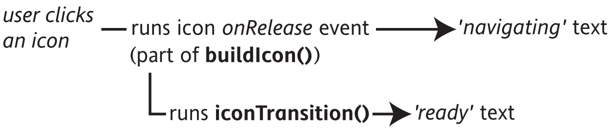
Looking at the diagrams of how the code works (in the section "A great big sanity check"), there are two ways to start a navigation and two ways to end a navigation:

- A navigation can start when the user clicks a strip—this triggers `navigate()`—or an icon, which triggers `buildIcon()`.
- A navigation can end at the end of `posTransition()` or `colTransition()`, depending on what kind of page the user is navigating to.
- So, you need to add a navigating message in `navigate()` and `buildIcon()`, before the forward/backward animations start. In addition, you need to return the message text to ready at the end of both `posTransition()` and `colTransition()`. You also need to return it to ready at the end of the `iconTransition()` event handler.

Again, a couple of diagrams should help summarize the concepts just described. The following illustration shows what you need for the forward navigation.



And this is how the back navigation works.



Let's implement this in the code, starting with the forward path.

Setting the status text for the forward path

1. The status text reads ready at the start of the site (because you've manually entered this into the status_txt text field). As soon as the user clicks a strip, you want to change the text to navigating. To do this, add the following new lines at the end of the function navigate():

```
  this.onEnterFrame = posTransition;
  // Set status text
  status_txt.text = "navigating...";
}
```

2. When the navigation has completed, the text needs to return to ready. Add the following code to the last part of posTransition() (in the final else statement):

```
    } else {
      // otherwise place the content
      placeContent();
      // and set status to 'ready'
      status_txt.text = "ready";
    }
  }
}
```

3. If this else statement doesn't run, that means colTransition() has been called instead, so you need to add the following line to the last else statement in colTransition():

```
  } else {
    // we are now at the new page
    color0 = tricolor.colStrip0.getRGB();
    color1 = tricolor.colStrip1.getRGB();
    color2 = tricolor.colStrip2.getRGB();
    fadeText(100);
    delete this.onEnterFrame;
    populateEvents(currentPage);
    status_txt.text = "ready";
  }
}
```

This ensures that the status text will be updated correctly regardless of whether it's a content page or a link page.

Setting the status text for the backward path

1. If the user clicks an icon, you need to make the text read navigating. The code that runs when the user clicks an icon is the onRelease event handler defined within buildIcon() (it's anonymous, so it has no name). To do this, add the following to the end of buildIcon():

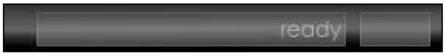
```
    this.onEnterFrame = iconTransition;
    // set status to 'navigating'
    status_txt.text = "navigating...";
  };
}
```

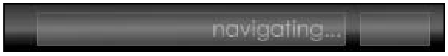
2. You want to return the text to ready at the end of the onEnterFrame event handler that controls the icon transition from "little icon" to "full screen." This occurs at the end of the last if statement in iconTransition() (which runs on the last onEnterFrame event handler in the animation). Add this to the bottom of iconTransition():

```
    // update the icon count and
    // remove this icon
    iconCount = this.iconCount-1;
    this.removeMovieClip();
    // update status text
    status_txt.text = "ready";
  }
}
```

The status text will now switch between ready and navigating when the interface is running. Not the biggest feature of the UI, but it does have one very big advantage. In placing these text changes into the code, you've been forced to understand how the (nearly complete) code hangs together, and this is, after all, a teaching exercise!

> *The status text actually started out as a bit of diagnostic text to check what was happening to the UI as it ran. It seemed a cool feature for showing what the code is doing at any time so it became part of the design, as shown in the following screenshots.*





There's a second text field that you haven't looked at: the statusValue_txt text field. In the final site, this gives you numeric feedback, such as the percent loaded during content load operations. You can't use this text field just yet, because the site isn't capable of such operations.

You're almost there. You have one final thing to do before the UI is finished and you can start adding the final content.

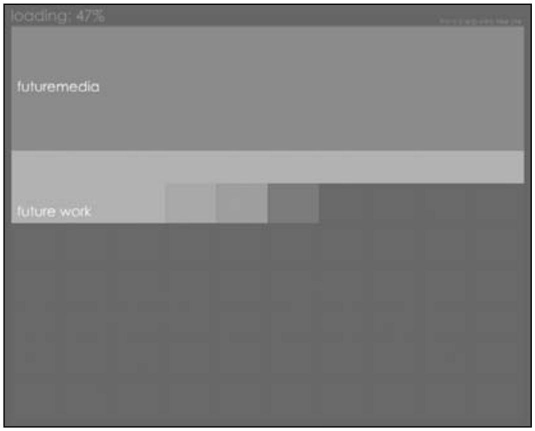
Adding a preloader

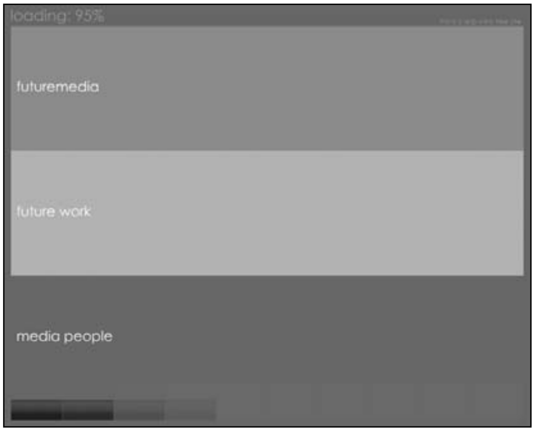
The site so far looks pretty, but you have no idea what the final content will look like or how bandwidth heavy it will be. Without a preloading strategy, you'll be stuck if this content turns out to be bandwidth heavy. Your entire site currently lives on frame 1 of the timeline, which is bad news if the site has to load a significant amount of content on that frame, since the user will see a blank screen until all the content is loaded and the first frame of Futuremedia can be shown. You have two ways around this:

- You can create a preloader within the current FLA, perhaps using the `ProgressBar` component.
- You can create a separate preloader file.

Although the `ProgressBar` component gives you a professional-looking preloader with only a few lines of code (see Help ➤ Flash Help ➤ Components Language Reference ➤ ProgressBar Component), the first option isn't recommended, because you don't know how much content will have to be loaded in frame 1. This is particularly true if some of your content requires the loading of class definitions, which are always loaded at frame 1 by default. You can change this, of course, but it's better to cater for the default.

The preloader fades in the site as a series of colored blocks while content is loading. The percentage loaded is also listed at the top left via a text message, as shown in the following screenshots.





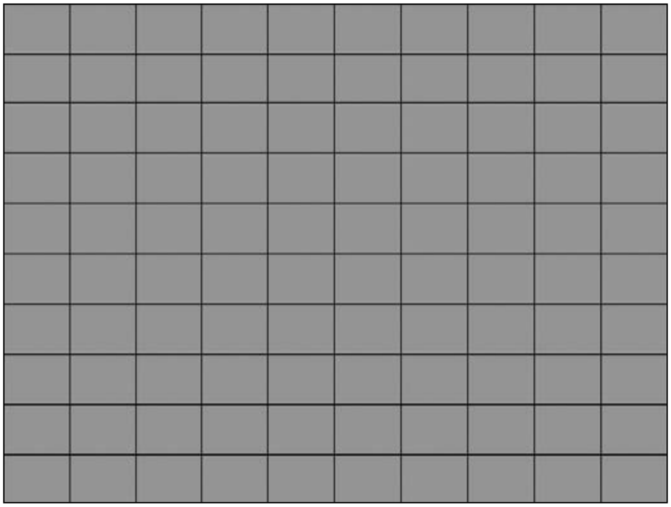
Of course, if you have a high-bandwidth connection, you'll miss all of this. That's actually a good thing—nobody wants to wait for long downloads, so the site tries to make the best of it by giving the low-bandwidth users something to look at. Be wary of making your preloader or start screen too long. A cool but long intro says you expect users to visit this site only once. Even worse are intro screens that are so big they need a preloader of their own!

Have a look at index.fla in the download files for this chapter. This is your preloader. You may have been wondering how the final site loads the tricolor as a series of fading blocks, when the tricolor isn't made out of blocks; it's a single complete graphic. This is how it's done—as usual, you cheat. You actually cheat big-time, because everything you assume about the preloader is incorrect.

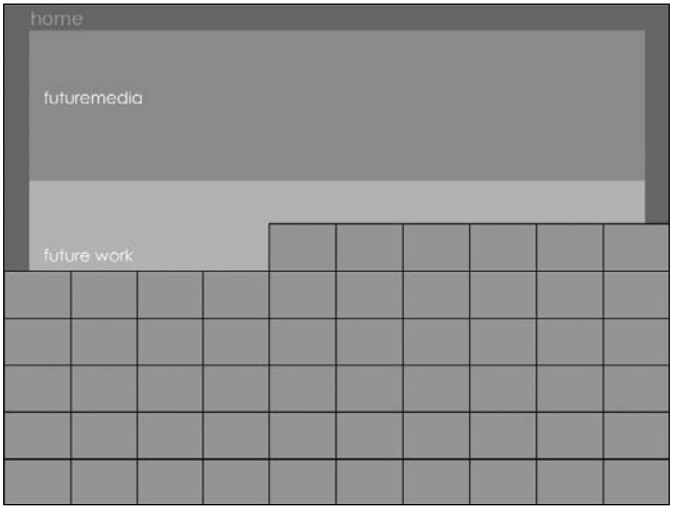
- The site that appears slowly during the preload is *not* the Futuremedia site—it's a copy.
- The site that does appear during the preload doesn't load over time—it loads *immediately*. You simply reveal it over time to give the impression that it's loading over time.

When you open index.fla, you'll see something that looks exactly like the home page of the Futuremedia site. That's it, though; the tricolor is just some static graphics, and nothing on this version of the site actually works. Because it does nothing, this doppelganger has one very cool feature: it will load almost instantaneously.

When index.fla loads, it's hidden behind a set of blocks that are created dynamically. These are set up as a grid. At this point, the code tells Flash Player to start loading the main (working) site that you've been building so far.



As loading of the real site progresses, the blocks are removed to reveal the nonfunctioning doppelganger. The blocks are removed by code that looks at the percentage loaded of the main site (the real site is hidden at this point, using yet another sneaky and totally obvious-once-you-know-it technique we'll reveal in a moment), and the code removes the same number of blocks (remembering that you have 10 × 10 blocks = 100 blocks total, so each block can be equated to 1% loaded).



Because the blocks are very close to the background color, it doesn't look like the blocks are disappearing, but more like the site is *appearing*.

Once all the blocks have loaded, you have your dummy site fully revealed. The trouble is that it's a turkey. It doesn't work.

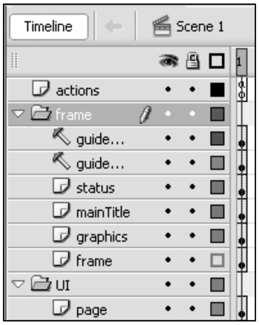
But wait! You have a fully revealed site only when the *real site is 100% loaded*. So here's what you do: you *switch the two sites*. The nonworking dummy is discarded, and the real site is made to appear in its place. The user will see none of this—the user will think it was the real site loading all along.

We'll leave you to have a look at the code in the `index.fla` file, but the following pointers should aid your understanding of it:

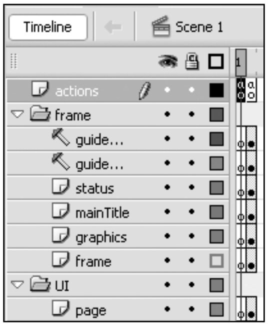
- The fading effect is *not* caused by the site fading in, but by the individual blocks fading *out*. You can see this if you look in the Library for `index.fla` and examine the `mc.block` clip. It consists of a simple tween that goes from 100% alpha to 0% alpha. By stopping all the blocks at the start and running each in turn, you achieve the fade-in effect.
- In the code, `_level0` refers to the preloader SWF, and `_level1` refers to the main (real) Futuremedia site SWF.
- The last thing the preloader does is a `_level1.play()`. This causes the by now fully loaded Futuremedia site timeline to start playing.

All you need to do now is that "hide the real Futuremedia site" trick. The clue to your final bit of magician's misdirection is in the last bullet point. You'll add a new keyframe to the main site, and leave this empty and with a `stop()` action on it. While the main site loads, it's stuck on this blank keyframe and therefore doesn't show up. The code in `index.fla` makes the main site play when the main site is fully loaded, and it's this action that reveals the real site.

The main timeline in futuremedia_code05.fla looks like this:

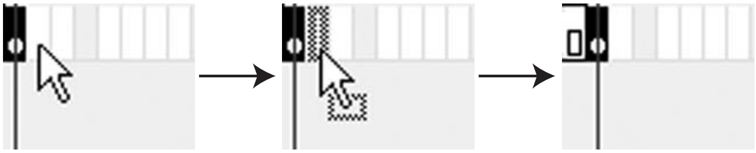


The final site file, main.fla, looks like this:



The only differences are as follows:

- All the keyframes have been moved from frame 1 to frame 2. You can do this to your own work-in-progress file by selecting each keyframe currently on frame 1 (first image), and then click-dragging it (second image) and dropping it over frame 2 (third image).



- The new frame 1 has a single stop() in the new empty frame 1 (layer actions).

If you run main.fla on its own, you'll see a blank stage. Right-click/Ctrl-click the blank stage of a running SWF and select Play to force the site to go to frame 2. Alternatively, run index.fla after compiling main.swf (i.e., testing main.swf at least once to create a SWF).

And that's it. You now have a fully functioning UI. What you *don't* have is the content to populate it. In the downloads for this book, you'll find one final ZIP file, Futuremedia_complete.zip. This contains an example of the type of material you could incorporate into the navigation system. Take a look at it, study the code, and then experiment with content of your own

Loading your own content into the site

Adapting the Futuremedia site to house your own content requires very little extra work—apart from creating the content, that is!

First of all, you need to create an XML document that contains the site's structure. Follow the same pattern as futuremedia.xml. The root node should contain three child nodes, each of which can have a maximum of three child nodes of its own. The deepest you should go is four levels, by which time you should have reached a content page. Refer back to the previous chapter if you need to refresh your memory about creating an XML document.

The tag for a content page (<contentPage>) will normally contain an id attribute or a url attribute. Use the id attribute for movie clips that will be loaded from the Library, and give the id the same name as the movie clip's linkage identifier. Use the url attribute for movie clips that you want to load at runtime.

You need to adapt the placeContent() function to load your own content instead of the placeholder movie clip. Change it like this:

```
function placeContent():Void {
  if (page[currentPage].pID != null && page[currentPage].pID != "") {
    _root.attachMovie(page[currentPage].pID, "content_mc", 10000);
    _root.content_mc._x = BORDER;
    _root.content_mc._y = BORDER;
  } else if (page[currentPage].pURL != null && page[currentPage].pURL ➥
!= "") {
    var mc:MovieClip = _root.createEmptyMovieClip("content_mc", 10000);
    var mcLoader:MovieClipLoader = new MovieClipLoader();
    var mclListener:Object = new Object();
    mcLoader.addListener(mclListener);
    mc._x = BORDER;
    mc._y = BORDER;
    mcLoader.loadClip(page[currentPage].pURL, mc);
    mclListener.onLoadInit = function(target_mc:MovieClip) {
      target_mc._width = Stage.width-BORDER*2;
      target_mc._height = Stage.height-BORDER*2;
    };
  }
}
```

This function checks the value of page[currentPage].pID and page[currentPage].pURL. If the pID is not null or an empty string, its value is used as the linkage identifier to load a movie clip from your Library into the content page. If the if statement fails, it then checks the value of pURL for the current page. If a suitable value is found, the function creates an empty movie clip and then uses the MovieClipLoader class to load the content from the relevant URL.

You can use a mixture of id and url attributes in your XML file to load content into your site. However, if a particular content page has values for both the id and url attributes, the placeContent() function gives priority to the id. The best way is to give each content page *one* of the attributes, but not both. Because navigationData() sets default values for every page, the missing attribute will automatically be set to null.

If your external files are likely to be large, you will probably want to add a preloader script to the else if statement. By now, though, we expect you have sufficient knowledge of ActionScript to be able to do that yourself.

Parting shots

There must have been times when you wondered if you would ever finish this case study, or even if it was worth doing so. Even if you haven't typed in all the code yourself, we hope that you've worked through the download files to get a better understanding of how a medium-sized project like this is put together. The case study has covered a lot of ground—and not all of it has been easy.

What we hope you've learned from this experience is that one of the major differences—if not *the* difference—between a Flash beginner and an ActionScript master is knowing how to keep Flash's graphics, code, and data environments totally separate. Not only is the content stored in external files, but also the navigation data for the Futuremedia site isn't hard-coded into the timelines. It isn't even contained in the code. It's in a separate file altogether—one that adds a completely new dimension to your Flash development: the ability to create sites where the content is dynamically driven by external data.

That's the technical message of the Futuremedia site, but there's a more philosophical issue to think about as well. Notice that the site looked really unfinished for almost the entire book, but now, right at the end, it has all come together *quickly*. Simply add two lines, and every page starts to show titles. Add a couple more lines, and the navigation moves forward in massive leaps toward completion. The reason the project went this way is because of what went into it: careful planning. We knew how the site should work right at the beginning, created copious sketches and tests, and stuck with it through all the "nothing much happening" stages, because we knew what to look for behind the scenes: the way the data was changing even though the graphics were not budging.

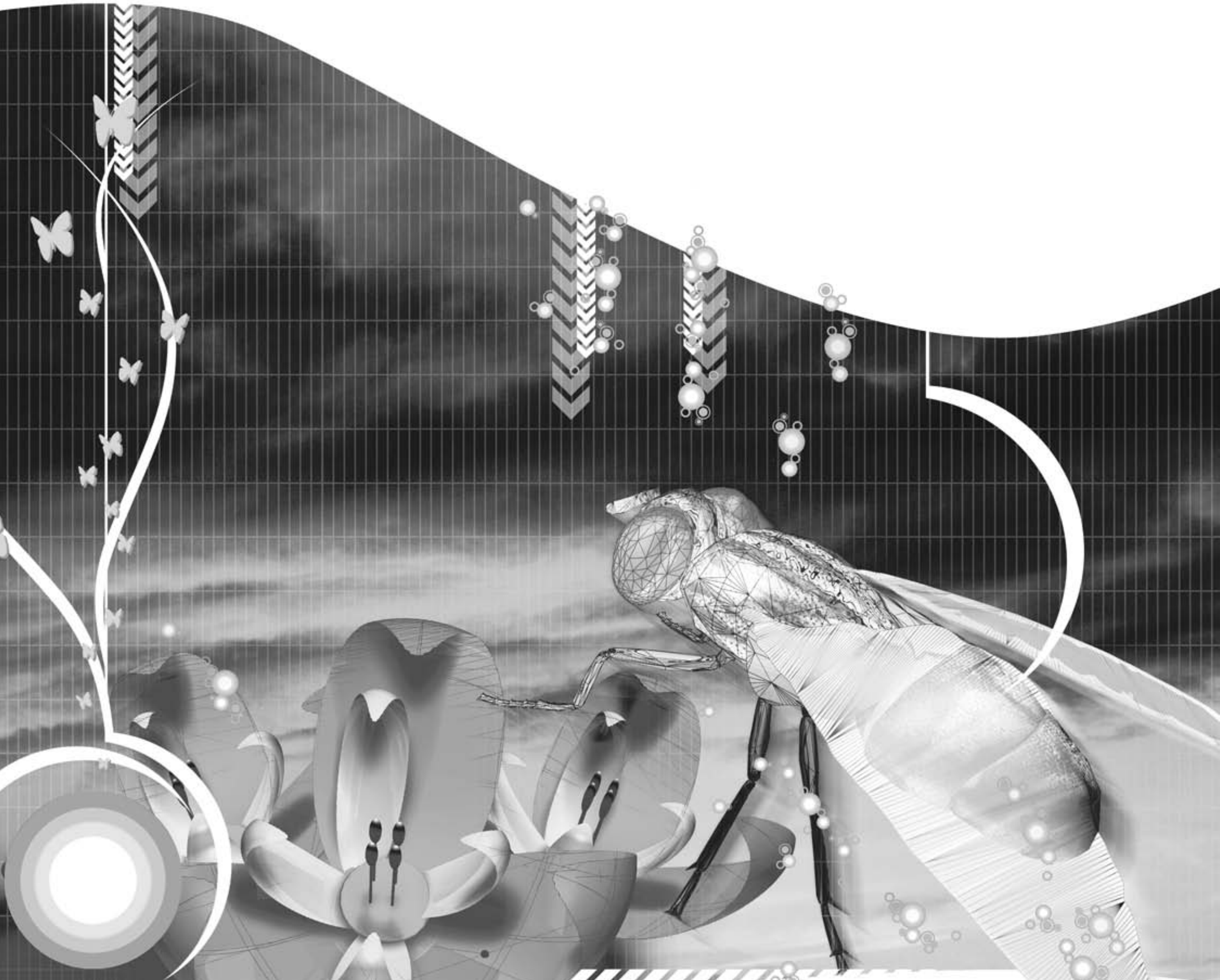
The difference between a senior programmer and a novice is the level of planning the senior programmer does and the confidence the senior programmer places in the planning. It can take a long time before you actually start to see stuff begin to move the way you want it to in a complex interface. Without a high level of confidence, you'll always be stuck with simple timeline sites that show results immediately.

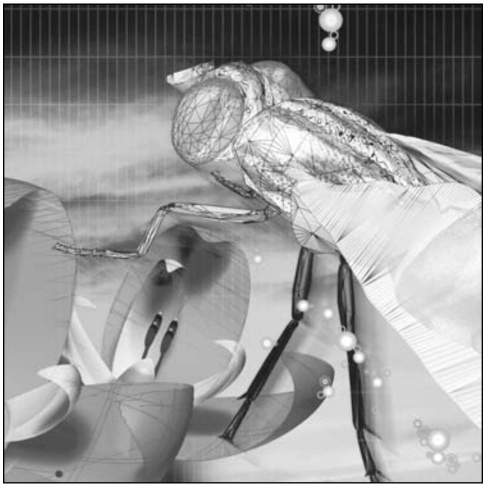
Summary

Don’t worry if you’re thinking, “Heck, I just couldn’t keep going this long! I would have given up well before this point.” We’ll let you in on a secret. The Futuremedia site design didn’t always go as smoothly as this book implies. Although it was originally designed by Sham, each of the authors has contributed to the final design. A few times we had to go back a number of steps to fix a small problem. Getting the XML structure right for this chapter involved writing a much longer script to make sure that everything worked. Then it was continuously refined and made more efficient. The things that kept the whole thing moving were the fact that the code is well structured (so it’s easy to modify even though there’s lots of it) and the fact that we knew it would work because our planning said it should. That’s partly a result of that “confidence” word again.

Rest assured that confidence comes more quickly than you think—it’s a bit like riding a bicycle. The first time you get it right with your own site design, you’ll have that confidence to keep it going straight without losing your balance. And like riding a bicycle, once that happens, you’ll have it for life.

Just one more chapter to go. It covers two aspects of advanced ActionScript: using version 2 components and building your own ActionScript classes. At this stage in your development, though, don’t be put off by the word “advanced.” You should be able to take both subjects in your stride. In fact, by now we think you’ll probably find them quite easy.





Chapter 15

ADVANCED ACTIONSCRIPT: COMPONENTS AND CLASSES

What we'll cover in this chapter:

- Using Flash components and the listener event model
- Extracting data from the `RadioButton`, `CheckBox`, and `ComboBox` components
- Extending ActionScript with classes
- Setting up a classpath

So far in this book you've created a lot of Flash content that relies on ActionScript. You've done some clever things with a very small subset of the ActionScript language, but there will come a time when you need to go further. At the moment, you're taking Flash building blocks such as movie clips and using them to create your content. How about creating some new building blocks of your own?

You can add new classes to Flash or extend some of the existing classes such that the available methods and properties are better geared toward solving your problem. In one respect, the Flash development team has already done this "extending ActionScript with new classes" deal by creating **version 2 components**. These are typically a series of self-contained movie clips that allow for the quick creation of common UIs and other cool stuff—everything from components that create scrollbars to components that define totally new software classes that are "added" to ActionScript.

Flash version 2 components

In Chapter 9, you built some simple components to control other movie clips. Useful though such components are, the built-in components that you'll find in the Components panel (Window ➤ Components) are in a completely different league. Flash version 2 components first made an appearance in Flash MX 2004 and use a completely different internal structure from the simple components that shipped with Flash MX (normally referred to as **MX UI components** or **version 1 components**).

This different structure posed quite a few problems for developers:

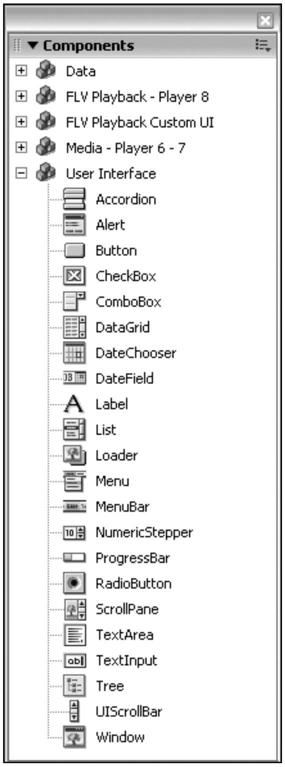
- You can't mix the two types of components in the same movie.
- Adding a component can increase the size of your SWF by as much as 50KB.
- Version 2 components require a minimum of Flash Player 6.
- Version 2 components use a different event model, which many found confusing.

Hmm. That makes version 2 components sound like a bad deal. In fact, they're quite the opposite, and now that you've reached this level of ActionScript, the different event model is something that you should be able to take in stride. What's more, a minimum of Flash Player 6 is now virtually standard, so there's no need to use the old components. The extra kilobytes added to your SWF are also a minor consideration, particularly if you use several components in the same movie. One component increases file size by 25–50KB, but adding extra components increases file size by only a few kilobytes. This is because version 2 components make extensive use of common classes.

The great advantage of version 2 components is that they add the sort of sophisticated functionality to your UI that would take ages to build yourself. Internally, they are very complicated, but all that complexity is hidden from view. They're **precompiled,** which means that you can't open them to take a peek inside. This makes the component a perfect black box.

You can see the components that ship with Flash 8 by looking in the Components panel (Window ➤ Components). The components available to you depends on the version of Flash that you're using. The screenshot at the top of the facing page shows the impressive range that comes with Flash Professional 8. The `Data`, `FLV Playback`, and `Media` components are not available in Flash Basic 8, and there are eight fewer `User Interface` components, too. Still, even Flash Basic users have an impressive range of components to enhance their movies, including the most frequently used, such as `CheckBox`, `ComboBox`, `RadioButton`, `TextArea`, `TextInput`, and `UIScrollBar`.

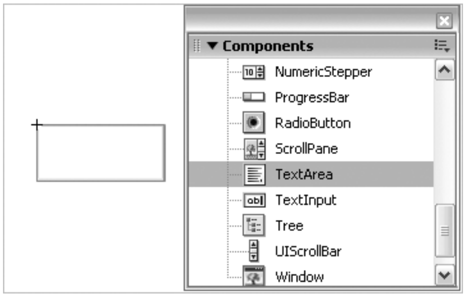
Because Flash is a visual medium, you can do some basic customization of components using tools to position and size them on the stage, as well as setting some parameters in the Property inspector. To make the best use of components, however, requires a good knowledge of ActionScript. Let's start off with some simple customization.



Working with a component on the stage

So far in this book, you've used text fields. They're great but they're a little basic. They don't have a lot of the niceties of text fields in commercial applications, such as scrollbars and the like. If you wanted to build that sort of thing, you *could* build it by yourself, but why go to all that trouble?

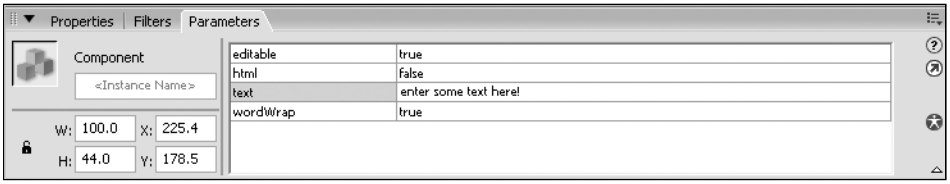
1. In a new blank document, open the Components panel (Window ➤ Components). Find TextArea (it's in User Interface), and drag it onto the stage in the same way as you would drag a movie clip from the Library. You'll see a little rectangle appear on the stage. This is an instance of the TextArea component.



You can also place a component onto the stage by double-clicking the component in the Components *panel.*

You'll see a new symbol appear in the Library panel. Note that its Type is listed as Compiled Clip. You can't edit it or view its timeline, no matter what you do!

2. Keep the rectangle selected. In the Property inspector, click the Parameters tab. You'll see a limited number of parameters for the component, as shown in the following screenshot. Select the text field, and type the following in it: enter some text here!



Component

<Instance Name>

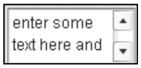
W: 100.0 X: 225.4

H: 44.0 Y: 178.5

editable	true
html	false
text	enter some text here!
wordWrap	true

As soon as you press Enter/Return or deselect the text field, you'll see the component reflect the changes in its parameters. Flash version 2 components are WYSIWYG ("what you see is what you get"), and unlike standard movie clips, they can sometimes start working as soon as they're on the stage (rather than when they're part of a running SWF).

3. Change the text to enter some text here and see what happens!

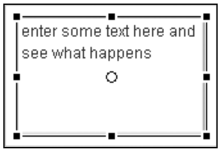

enter some text here and

After you have typed in the extra text, click anywhere in the stage to deselect the TextArea component. It will change to reflect that the new text needs scrollbars. The scrollbars won't actually work just yet, but you have a good idea of how the final component will look with your chosen text entered in it.

4. You would be better off with a larger text area. Select the TextArea instance on the stage, and then select the Free Transform tool.

You can now resize the TextArea component to whatever shape and size you want. If you did this with an ordinary text field, the text would get larger, and the scrollbar would still be there. With the component, though, only the area that the text is displayed in changes size, while the text remains unchanged and reflows into the new available space. You may also lose the scrollbar if it's no longer needed. (This part of the display of components isn't always WYSIWYG. You sometimes need to check by testing the movie.)

Although you can also use the W and H fields in the Property inspector to resize the component, this is not recommended, as it may distort some of the contents. The other way to change the size is with ActionScript, as you'll see shortly.

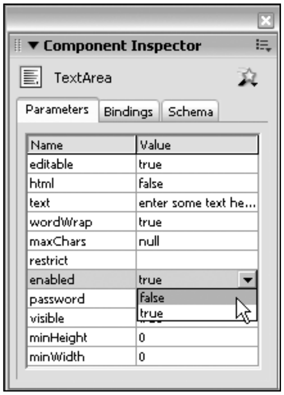

enter some text here and see what happens

5. Test the movie, and type some more text into the text area. As soon as the scrollbar is needed, it will appear automatically, making the input or display of large areas of text much easier than with a regular text field!

6. Although the Property inspector gives you *some* control over the component, it doesn't list all available parameters. To access more parameters, you need to look at the Component Inspector panel (Window ➤ Component Inspector).

   Open the Component Inspector panel and select the TextArea instance on the stage. The Parameters tab shows you more options. As an example, set the enabled parameter to false, as shown.

7. Test the movie again. The TextArea component is grayed out (it isn't enabled). Keep the FLA open, as you'll need it again in the next exercise.



Name	Value
editable	true
html	false
text	enter some text he...
wordWrap	true
maxChars	null
restrict	
enabled	true
password	false
visible	true
minHeight	0
minWidth	0

So, as you can see, version 2 components offer some useful features, such as the automatic scrollbar in the TextArea component and the ability to resize the components with the Free Transform tool. But that's only the tip of the iceberg.

Getting more out of components with ActionScript

Working with components like this is OK for beginners, but you can do far more with ActionScript. For example, rather than manually place the text area onto the stage, what if you could dynamically attach instances of the TextArea component onto the stage, so that you could create interactive forms? You could do all that resizing and other stuff you entered into the Property inspector at runtime via code and gray out TextArea instances on a form unless they're actually appropriate. With ActionScript you can do the following:

- Change the way the component instance *works or looks* by changing its properties.
- Make the component *do things* by using component methods.
- Make the component instance *respond to interaction* and other events by defining event handlers.

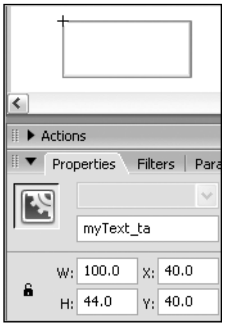
The Flash development team has done its utmost with the release of Flash 8 to make as much information as possible available to developers about working with components. In fact, the team may have done too much. The Flash Help files now contain two sections devoted to components: Using Components and Components Language Reference. They're also available as PDF files from www.macromedia.com/support/documentation/en/flash/. The problem is that they amount to more than 1,700 pages. Don't let that put you off, though. The bulk of the documentation is taken up by the Components Language Reference, which you should treat as a good dictionary. The use of each method and property is clearly described, and most entries are accompanied by a short example that shows you exactly how to use whatever it is you're looking up. Both the Flash Help files and the PDF equivalents are fully searchable, so once you find your way around them, you should make rapid progress.

The cool thing about components is that once you know how to work with one or two of them, learning to use the others is usually pretty easy, since many methods and properties are common to all of them.

You change component properties in much the same way as with movie clips: simply use dot notation. Let's see this in action.

Configuring a TextArea component with ActionScript

1. You need to work with a new instance of the TextArea component, so delete the current instance and drag a new one onto the stage—this time from the Library panel. Using the Property inspector, give the component an instance name of myText_ta.

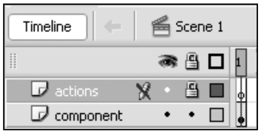


myText_ta

W: 100.0 X: 40.0
H: 44.0 Y: 40.0

When you drag a component from the Components *panel to the stage, Flash imports the component into the Library. You should therefore drag subsequent components of the same type from the Library. If you don't want to use a component in a FLA anymore, be sure to delete it from the Library as well as from the stage. Otherwise, the component will be exported in the final SWF, regardless of whether it's actually needed, unnecessarily bloating your file size.*

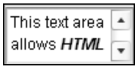
2. Rename the current layer component. Add a new layer called actions and lock it.



3. In the first frame of actions, insert the following script:

```
myText_ta.html = true;
myText_ta.editable = true;
myText_ta.text = "This text area allows <b><i>HTML formatting</i></b>";
```

This makes the TextArea instance display HTML text, makes it editable, and adds some HTML-formatted text inside it. If you test the movie, you'll see something like the screenshot alongside.

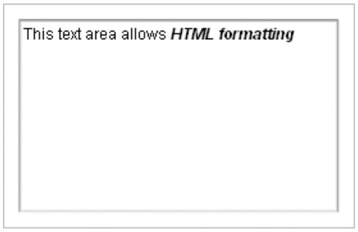


This text area allows ***HTML***

Notice that with a TextArea component, when html is set to true you can set HTML text directly using the text property. With a normal text field, you would need to specify htmlText when assigning a display string.

4. Cool, except it's small. So how do you make it larger? The natural assumption is that you're dealing with a text field, so using the _height and _width properties of the TextField class should do the trick. Wrong. The way to resize components is with the setSize() method, like this:

```
myText_ta.html = true;
myText_ta.editable = true;
myText_ta.text = "This text area allows <b><i>HTML formatting</i></b>";
myText_ta.setSize(250, 150);
```



This text area allows ***HTML formatting***

The setSize() method works for all version 2 components. It normally takes two arguments: *width* and *height*, in that order. When working with some components, such as the ComboBox component, you can supply just one argument to set the width. This doesn't work with TextArea—you must set both width and height.

Keep this FLA open because you'll need it again for the next exercise.

> *When working with components, you should never communicate with them using movie clip properties. Instead, use the properties and methods of the individual component class and of* UIObject, *a special class that controls the styles, events, and resizing of all components. You can find details of all the* UIObject *methods and properties in the Components Language Reference.*

At the moment, you have something that works well graphically, but you have no efficient way to work with the text that the user inputs into the TextArea. For that, you need to know how to set up your component to respond to events.

Components and event handling

Components use a more advanced version of the event model you've looked at so far for movie clips and buttons. They use **listeners**, which listen for **broadcast events**.

How event listeners work

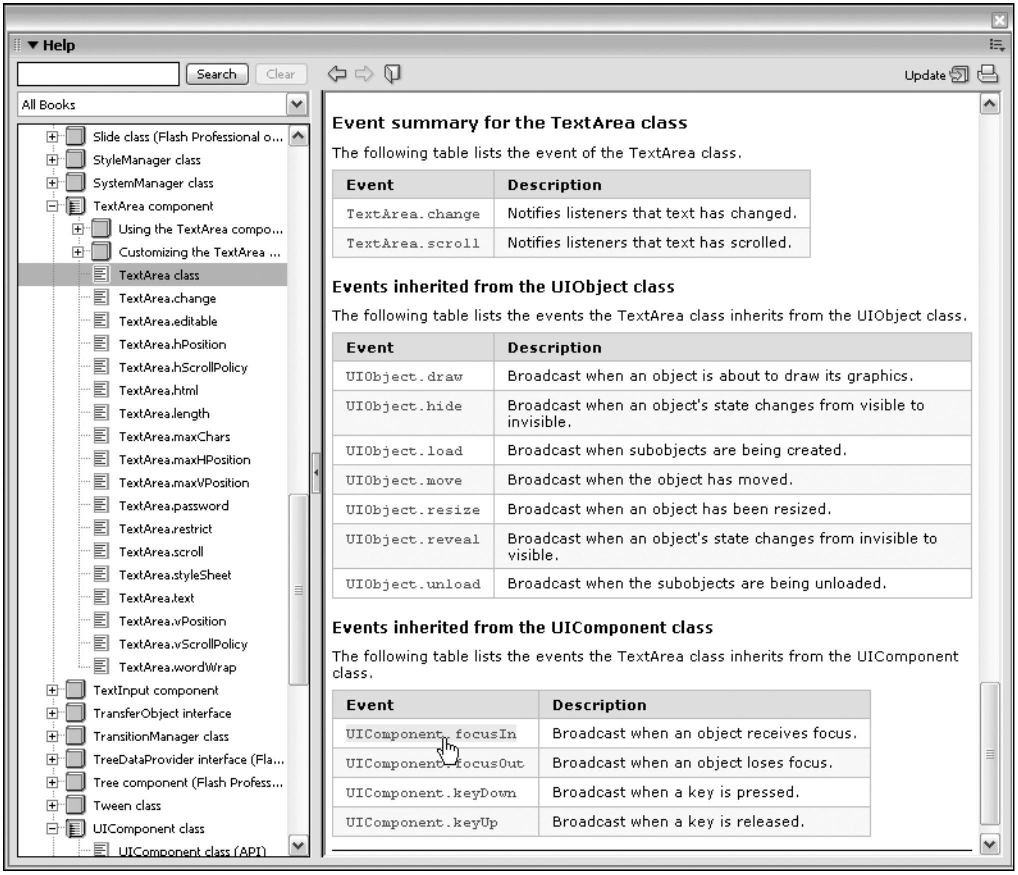
Rather than the event/event handler pair you've looked at so far, a listener-based event model consists of a **broadcaster** and **listeners**. A normal event/event handler pair is rather like a telephone communication. There's only one recipient: the event handler on the other side of the connection. So, if a button is clicked, there can be only one function that acts as the event handler.

The event model used by components is more like a radio broadcast. The event is broadcast everywhere. The number of recipients isn't just one person, but everyone who is listening in.

When setting up a listener, you do the following:

1. Set up a new object to listen.
2. Attach event handlers to the new object.
3. For each event you want to listen to, define the event handler you want it to be linked to.

The events that you can use with a component are different from the onRelease, onMouseDown, and other events that you've been using with movie clips and buttons. Each component has a class of its own, and the available events are listed in the class summary, which you can find by opening Help ➤ Flash Help ➤ Components Language Reference, expanding the name of the component, and selecting its class. The following screenshot shows the events listed for the TextArea class and component.



Event summary for the TextArea class

The following table lists the event of the TextArea class.

Event	Description
`TextArea.change`	Notifies listeners that text has changed.
`TextArea.scroll`	Notifies listeners that text has scrolled.

Events inherited from the UIObject class

The following table lists the events the TextArea class inherits from the UIObject class.

Event	Description
`UIObject.draw`	Broadcast when an object is about to draw its graphics.
`UIObject.hide`	Broadcast when an object's state changes from visible to invisible.
`UIObject.load`	Broadcast when subobjects are being created.
`UIObject.move`	Broadcast when the object has moved.
`UIObject.resize`	Broadcast when an object has been resized.
`UIObject.reveal`	Broadcast when an object's state changes from invisible to visible.
`UIObject.unload`	Broadcast when the subobjects are being unloaded.

Events inherited from the UIComponent class

The following table lists the events the TextArea class inherits from the UIComponent class.

Event	Description
`UIComponent.focusIn`	Broadcast when an object receives focus.
`UIComponent.focusOut`	Broadcast when an object loses focus.
`UIComponent.keyDown`	Broadcast when a key is pressed.
`UIComponent.keyUp`	Broadcast when a key is released.

Some events are exclusive to a particular component. Others are shared with other components and inherited from either the `UIObject` or `UIComponent` class. The Flash 8 Help files make it easy to find the necessary information. Just click the event name, and you'll be taken to the relevant help page. You can also use the left arrow at the top of the Help panel to return to the page you've just come from.

That's a lot of information to take in without an example to back it up. Let's fix that `TextArea` from the last exercise with some listener-based event handling before exploring the listener event model in general.

Clearing text from a TextArea component

In this exercise, you'll add a listener to the `TextArea` component, so that when you click inside it the existing text will be cleared. Continue using the FLA from the previous exercise. You can find the finished code in `listener01.fla` in the download files for this chapter.

1. First, you need to add a new object that will become your listener. Edit the code, adding line 1 to create a new object instance, `textListener`, and changing the displayed text in the second-to-last line. Notice that `textListener` is just an ordinary object—you don't have to do anything special to it to make it a listener.

   ```
   var textListener:Object = new Object();
   myText_ta.html = true;
   myText_ta.editable = true;
   myText_ta.text = "Enter some text <i>here!</i>";
   myText_ta.setSize(250, 150);
   ```

2. You want something to happen as soon as the user clicks inside the `TextArea`. The event that deals with this is called `focusIn`. It's a general event for all UI components (rather than a specialized event for the `TextArea` class), so it comes from the `UIComponent` class. Change the code as shown here:

   ```
   function clearText():Void {
     trace("you clicked inside something!");
   }
   var textListener:Object = new Object();
   textListener.focusIn = clearText;
   myText_ta.html = true;
   myText_ta.editable = true;
   myText_ta.text = "Enter some text <i>here!</i>";
   myText_ta.setSize(250, 150);
   ```

   Notice that we have created a named function and then assigned it to `textListener.focusIn`. You could also use an anonymous function like this:

   ```
   var textListener:Object = new Object();
   textListener.focusIn = function() {
     trace("you clicked inside something!");
   };
   ```

   However, using an anonymous function means you lose one of the great advantages of the broadcast/listener event model: reusability. An anonymous function is attached to a single event, whereas you can use a named function with as many events and listeners as you like. Note that if you use an anonymous function, it must come *after* you have declared the listener object, because it's attached directly to the object.

3. The `clearText()` function will trace a message every time it is run, but at the moment, it will never run because `textListener` isn't listening for any event broadcasts. To make it listen to the `focusIn` event for `myText_ta`, add the following line highlighted in bold:

   ```
   function clearText():Void {
     trace("you clicked inside something!");
   }
   var textListener:Object = new Object();
   textListener.focusIn = clearText;
   myText_ta.addEventListener("focusIn", textListener);
   ```

```
myText_ta.html = true;
myText_ta.editable = true;
myText_ta.text = "Enter some text <i>here!</i>";
myText_ta.setSize(250, 150);
```

This adds the event listener to the TextArea component. The syntax is a bit unusual, but it's explained in more detail shortly.

4. Test the movie. When you click inside the TextArea, you'll see the message you clicked inside something pop up in the Output panel.

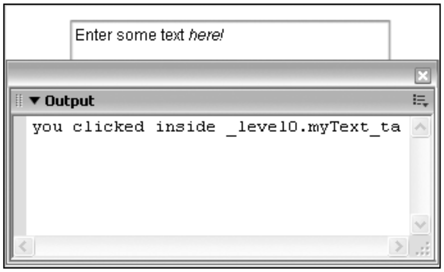
   If nothing happens, make sure that you're not still using the TextArea instance from the first exercise. The component's enabled property must be set to true.

   That's progress, but you actually want to do something with the particular component you clicked inside. Every event automatically generates an event object that contains information about the event that's just happened. By passing this object as an argument to your event handler function, you can extract that information for use inside the function. Change the clearText() function like this:

```
function clearText(evtObj:Object):Void {
  trace("you clicked inside " + evtObj.target);
}
```

The important thing to remember when you're using listeners is that you can't use your old friend this*, because the event handler isn't attached to the instance creating the event. It's attached to a listener object instead.*

5. Test the movie. The message will now tell you the component's name when you click inside it, because the target property of the event object informs you of the instance name of the broadcasting instance. By the way, you don't need to call the event object evtObj. Any name will do. (You can call it "sweetheart" if you like.)



Enter some text *here!*

```
you clicked inside _level0.myText_ta
```

6. Now that you know the instance name of the component that broadcast the event, you can use that information to clear the text inside it. Change the clearText() function as shown:

```
function clearText(evtObj:Object):Void {
  evtObj.target.text = "";
}
```

7. Test the movie again and click inside the component. This time, the existing text is cleared.

Adding an event listener

Now that you've seen the broadcaster/listener event model in action, let's go back over the elements that it involves. It's not difficult, but it takes a little getting used to, and since future versions of ActionScript are likely to go in this direction, it's important to get it right.

The main part of the script you have just created looks like this:

```
function clearText(evtObj:Object):Void {
  evtObj.target.text = "";
}
var textListener:Object = new Object();
textListener.focusIn = clearText;
myText_ta.addEventListener("focusIn", textListener);
```

The first three lines are the function that acts as the event handler. A broadcast event always generates an event object that contains information about the event and where it was triggered. If you pass this object to your event handler function, you can use it to extract that information. Although it's not always necessary to pass the event object as an argument, it's useful to know it's there if you need it.

Next, you create an object to act as the listener. In this example, we used the name `textListener`. It's a common convention to include `Listener` in the name, simply because it makes your code more understandable. However, there is no special "Listener" type. Just use an ordinary object.

So far, things are easy. It's the next two lines that seem to cause problems, so let's take them in turn:

```
textListener.focusIn = clearText;
```

The pattern here is as follows:

```
listenerObject.eventName = eventHandlerFunction;
```

This line must always come *after* you have defined the listener object, for the simple reason that the object must exist before you can assign any events to it.

Finally, you use `addEventListener()` to associate the event and the event handler function with the object you want to be on the alert for that particular event. In our example, it looks like this:

```
myText_ta.addEventListener("focusIn", textListener);
```

This means you want the `myText_ta` component instance to listen for the `focusIn` event, and when it "hears" it, to run the `focusIn` event handler for `textListener`. The pattern looks like this:

```
target.addEventListener("event", listenerObject);
```

The way you pass arguments to `addEventListener()` is rather counterintuitive, but that's the way it must be. Remember that the name of the event must be a string, so it needs to be in quotes.

Understanding the advantages of event listeners

Thinking back to the previous exercise, the chain of events inside the movie is as follows:

1. You click inside the TextArea instance myText_ta.
2. myText_ta broadcasts a focusIn event.
3. textListener is listening in to this broadcast and responds.

Compare this to what would have happened for a regular event/event handler pair:

1. Flash sees an event.
2. Flash looks to see if there is an event handler defined to handle this event.
3. If Flash finds an event handler defined to handle this event, it is executed.

The listener event model uses two instances, the broadcaster and the listener object, whereas the event/event handler pair requires only the event handler to be set up. The latter is an easier system to set up, but it's less flexible if you want to do something clever.

So what do we mean by "clever"? The beauty of listeners is that the event is *broadcast* rather than sent to a single handler only. This means you can have multiple listeners (among other things). Change the code in the FLA from the previous exercise as follows (or look at listener02.fla):

```
function clearText(evtObj:Object):Void {
  evtObj.target.text = "";
}
function fanfare():Void {
  trace("Ta-ra!!!!!");
}
var textListener:Object = new Object();
textListener.focusIn = clearText;
var soundListener:Object = new Object();
soundListener.focusIn = fanfare;
myText_ta.addEventListener("focusIn", textListener);
myText_ta.addEventListener("focusIn", soundListener);
myText_ta.html = true;
myText_ta.editable = true;
myText_ta.text = "Tell us why you want to <b>win $1,000!</b>";
myText_ta.setSize(250, 150);
```

This time, the chain of events is as follows:

1. You click inside the TextArea instance myText_ta.
2. myText_ta broadcasts its focusIn event.
3. textListener is listening and responds by sorting out the text.
4. soundListener is listening and responds by sorting out the sound. (Yes, we know there's no sound, but go on—just pretend there is.)

What's the point of that? Surely you could deal with both text and sound in the same function? Yes, you could, but just imagine if you're creating a really complex application for a client who suddenly decides the sound completely ruins the effect. All that you need to do is remove the `soundListener` and it's gone. No fiddling around with a function that may be handling several unrelated operations at the same time.

Listeners offer greater flexibility when compared with the restrictions of a single event/event handler pair:

- **The number of listeners per event is unlimited**. This means that you can set off *multiple event handlers* in response to some of the more important events. For example, suppose you put a "clear everything, I want to start again" button on a page. It makes sense to have everything listening to the "I have just been clicked" event of that one button, so that all the various UIs can be cleared or reset. With an event/event handler pair, the event handler would be the only thing to respond, something that makes the code for it difficult to manage (it has more to do). Instead, you can have lots of listeners looking out for the quit command and clearing their own specific part of the application.
- **Where the broadcaster is sending out multiple events, the listeners can choose to respond to only some events**. This is rather like a human listening to a radio station only when music from the alternative rock charts is being broadcast, because adult-oriented rock music and boy bands aren't suited to this particular music fan. In the same way, a particular Flash listener can choose to respond to those events that involve `TextArea` resizing only, because that's what it's written to handle. The advantage of this is that you get to *structure your event handling*.
- **A listener can respond to several different events**. In an event/event handler model, the event handler is limited in what it can respond to, because the link to available event sources is less flexible. A listener can respond to *any* broadcast event, because *all* of them are available to it; it can simply pick what it wants. This allows you to *generalize more*. A listener could be set up to listen to all events that would need the current SWF to quit. The "quitter" listener is thus not just listening for events, but more generally it's handling all events that lead to a particular outcome.
- **Listener-based events aren't attached to the instance broadcasting the event, which makes for better extensibility**. Adding a listener event doesn't affect the instance creating the event. This is very useful when you want to update code. Adding new listeners to a movie clip doesn't change what the movie clip does, but adding a new `onEnterFrame` deletes any `onEnterFrame` already attached to the movie clip.

Suppose you've been asked to build an online teaching application, which will create many text-based screens. To make the project easier, you've created several listeners, each looking at a particular type of event:

- An input listener, `inputManager`, which manages user input events
- A navigation listener, `navManager`, which manages user navigation events
- And so on

All your managers are *general* listeners that aren't written to be attached to *specific* instances, but *the general events they create*. Your advanced multimedia event-handling code is no longer a set of simple cause and effect systems, but a true data-broadcasting network. Events aren't one-way; rather, they're made open to any other code module that listens for them.

Commonly used components

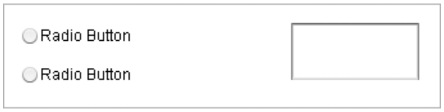
Components are particularly useful for building online forms. They provide a clean, professional, uniform look, so it's important to know how to get information into and out of them. Let's take a quick look at working with the RadioButton, CheckBox, and ComboBox components, which are frequently used in multiple-choice forms. The first two will probably be familiar to you from ordinary web pages. The ComboBox creates a drop-down menu, and the Flash version has a great advantage over its (X)HTML equivalent in that it can be configured to let users enter their own value if the choices offered aren't sufficient. We won't be covering every aspect of these components, but over the next few pages you should get a good insight into how they work.

Radio buttons

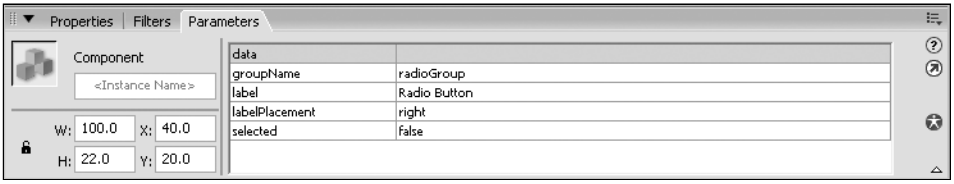
Radio buttons are ideal for a multiple-choice situation where the user is allowed to choose only one of several options: Yes, No, or Don't Know; Male or Female; that sort of thing. You can get them to display a particular selection by default, but once another button is clicked, the original one is deselected. You need to create an event handler function to find out which one.

Getting information from a radio button

1. Open a new Flash document and drag a RadioButton component onto the stage from the Components panel. Also drag a TextArea component onto the stage. If you just want to look at the code, you'll find the finished version in components01.fla.
2. From the Library panel, drag another instance of the RadioButton component onto the stage. Lay out the three components as shown here.



3. In the Property inspector, give the TextArea component an instance name of output_ta. You don't need to give the RadioButton components instance names, as they'll be identified in a different way.
4. Select the first RadioButton component on the stage, and then open the Parameters tab of the Property inspector, which should look like the next screenshot.



Component

<Instance Name>

W: 100.0 X: 40.0

H: 22.0 Y: 20.0

data	
groupName	radioGroup
label	Radio Button
labelPlacement	right
selected	false

This lets you set five options for the component:

- data: This is the value you want the radio button to have. Click inside this option and type y.
- groupName: This is the name that identifies all radio buttons that belong to the same group. If you have only one set of radio buttons in your movie, you can leave it at the default, radioGroup. For the purposes of this exercise, though, change it to agree.
- label: This is the label that appears on the stage alongside the radio button. Click inside this option and type Yes.
- labelPlacement: This gives you the option to place the label to the left, right, top, or bottom of the radio button. Leave it at the default right.
- selected: This determines whether the radio button is selected when the movie first runs. Leave it at the default false.

5. Select the second `RadioButton` component. Set the data field to n and the label field to No. Change groupName to agree, and leave the other two options at their default values.

6. Rename the current layer as components. Add a new layer called actions and lock it. Select the actions layer, open the Actions panel, and pin it on frame 1.

7. Type the following script in the Actions panel:

```
function queryRadio():Void {
  output_ta.text += agree.selection.label + " was selected\n";
}
```

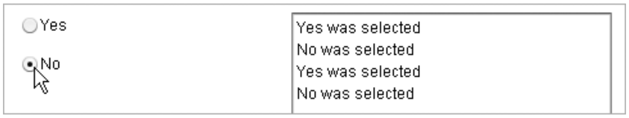
This will be the event handler function for the radio button group. It adds the value of `agree.selection.label` plus the string `"was selected\n"` (which ends with a newline character) to the `text` property of the `TextArea` component instance called `output_ta`. The way you identify which radio button has been selected is with the `selection` property of whatever you used as `groupName`. In this case, it's `agree`. By adding `label` to the end with dot notation, you get the value stored in the selected `RadioButton` instance's label.

You don't need to pass the event object to the `queryRadio()` *function because the function uses the* `groupName` *property to identify the correct radio button group.*

8. Before you can test it, you need to set up the event listener and assign the `queryRadio()` function to its `click` event. You also need to make the `TextArea` component bigger. Add the following code to the end of the script:

```
var agreeListener:Object = new Object();
agreeListener.click = queryRadio;
agree.addEventListener("click", agreeListener);
output_ta.setSize(250, 320);
output_ta.text = "";
```

9. Test the movie. Each time you click one of the radio buttons, the name of the button should appear in the text area alongside, as shown in the following screenshot.



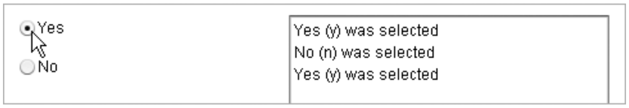
○ Yes
● No

```
Yes was selected
No was selected
Yes was selected
No was selected
```

10. Getting the value of the label is fine, but you often need to use a different value, particularly if the data from the form is going to be inserted into a database. That's why the RadioButton component also has the data property, which you access in exactly the same way as label. Change the queryRadio() function like this:

```
function queryRadio():Void {
  output_ta.text += agree.selection.label + " (" + ➥
agree.selection.data + ") was selected\n";
}
```

11. Test the movie again. You should now see the data value enclosed in parentheses after the label name, like this:



● Yes
○ No

```
Yes (y) was selected
No (n) was selected
Yes (y) was selected
```

12. Back inside the FLA, select the top RadioButton instance and in the Properties tab of the Property inspector, give it an instance name of yes_rb. Open the Actions panel, make sure the script pane is still pinned to frame 1 of the actions layer, and add the following line to the bottom of the script:

```
yes_rb.selected = true;
```

13. When you test the movie again, you'll see that the Yes radio button is selected, but nothing will appear in the text area until you click the No radio button. Save the FLA, as you will need it in the next exercise.

> *When working with the* RadioButton *component, it's important to remember the difference between* selected *and* selection*. The* selected *property is used to set the state of a particular radio button. The* selection *property, on the other hand, is normally used to check which button of a particular group has been selected.*

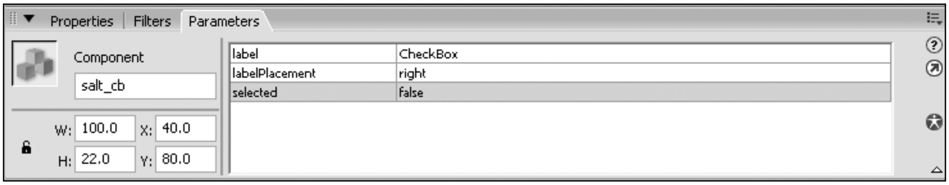
Check boxes

Check boxes are very similar to radio buttons, except that they're not mutually exclusive. You can select as many or as few check boxes as you like. This means that you need a slightly different way to determine the state of a check box.

Getting information from check boxes

Continue working with the FLA from the previous exercise. The completed code for this section is in `components02.fla`.

1. Open the Components panel, if it's not already open, and drag an instance of the CheckBox component onto the components layer on the stage. Then drag two more instances of the CheckBox component from the Library panel. Position the three new instances below the two radio buttons. In the Property inspector, give them instance names of salt_ch, pepper_ch, and vinegar_ch.
2. Highlight the first of the CheckBox instances and select the Parameters tab of the Property inspector. As the next screenshot shows, this time you have fewer choices.



Component
salt_cb
W: 100.0 X: 40.0
H: 22.0 Y: 80.0

label	CheckBox
labelPlacement	right
selected	false

3. Change the label field for each CheckBox component to match its instance name: Salt, Pepper, Vinegar.
4. Open the Actions panel on frame 1 of the actions layer and add the following script (strictly speaking, you should keep all functions together, but it doesn't matter for the purposes of this exercise):

```
function queryCheckbox(evtObj:Object):Void {
  var thisBox:Object = evtObj.target;
  if (thisBox.selected) {
    output_ta.text += thisBox.label + " was selected\n";
  } else {
    output_ta.text += thisBox.label + " was deselected\n";
  }
}
var checkboxListener:Object = new Object();
checkboxListener.click = queryCheckbox;
salt_ch.addEventListener("click", checkboxListener);
pepper_ch.addEventListener("click", checkboxListener);
vinegar_ch.addEventListener("click", checkboxListener);
```

The last five lines define the event listener, assign its `click` event to the `queryCheckbox()` function, and then add it to the three CheckBox instances. Nothing new there.

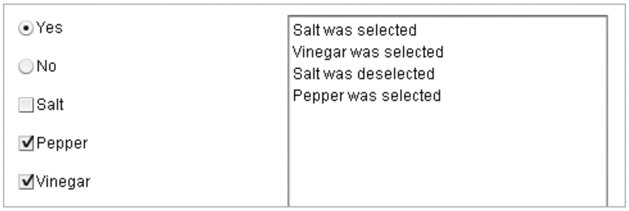
What you need to concentrate on here is the queryCheckbox() function. First of all, each CheckBox instance is treated individually by ActionScript—they're not identified as a single group in the same way as the RadioButton components or their (X)HTML equivalents. So you need to pass the automatically generated event object to the event handler function. Otherwise, the function has no way of knowing where the event was triggered. We've been pretty unimaginative and called it evtObj again, but it has the advantage of reminding us clearly what it's for.

To make things easier (and slightly more efficient) inside the function, we've assigned the value of evtObj.target to a local variable called thisBox. We then use the selected property of the CheckBox class to find out whether the component that was the target of the event has been selected.

> *Hang on a minute! With the* RadioButton *component,* selected *is used to set the state, but* selection *is used to check its existing state. The* CheckBox *component uses* selected *for both purposes. Hmm. Components are a bit trickier than at first sight.*

Depending on whether the CheckBox has been selected, the value of its label property is displayed in the text area with an appropriate message.

5. Test the movie, and try selecting and deselecting individual check boxes. You should see similar output to the following screenshot.



```
Salt was selected
Vinegar was selected
Salt was deselected
Pepper was selected
```

Note that although Yes is selected, it was set as the RadioButton group's default in the previous exercise, so nothing will be displayed for the RadioButton group unless you select No first. Your ActionScript is *listening for events*, not reflecting the current state of the components when the movie first loads. You should know the default state, anyway, because it's set by you when you create the movie or the script that runs it.

> *Because the* CheckBox *component doesn't have a* data *property, you can use only its* label *property or the* target *property of the event object to identify which check boxes have been selected.*

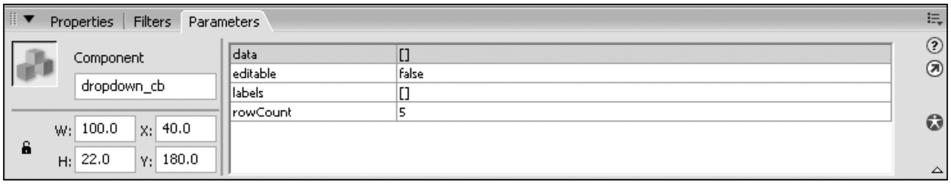
Combo boxes (drop-down menus)

The drop-down menu has become a familiar feature of web and software user interfaces. It enables you to offer many options without taking up valuable screen real estate. The ComboBox component is a quick and easy way to incorporate drop-down menus in Flash movies. Because it's so versatile, it has many more properties and methods than the other components we've looked at. These next two exercises will give you just a brief glimpse of the possibilities available.

Populating a ComboBox component with data

Continue working with the FLA from the previous exercise. The final code for this section can be found in `components03.fla`.

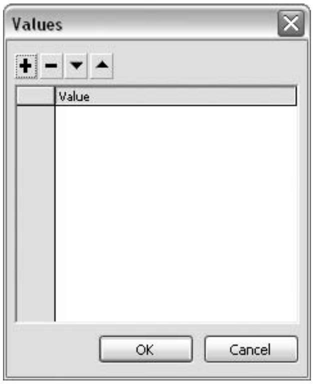
1. Drag an instance of the `ComboBox` component from the Components panel onto the components layer on the stage and give it an instance name of dropdown_cb. When you select the Parameters tab of the Property inspector, you will see the range of options shown in the following screenshot.



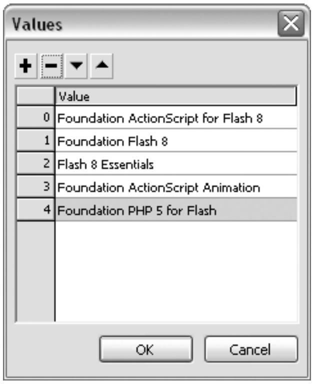
Component
dropdown_cb
W: 100.0 X: 40.0
H: 22.0 Y: 180.0

data	[]
editable	false
labels	[]
rowCount	5

2. The parameters are listed in alphabetical order, so it's the third item, labels, that you mainly need to concentrate on. As you might expect, the labels field determines the items that actually appear in your drop-down menu. In its default state, it contains just a pair of square brackets, which indicates that it's an empty array. Either double-click this field or highlight it and click the magnifying glass icon that appears at the right end. This brings up the Values dialog box, as shown in the screenshot alongside. This is where you enter the labels that you want to display in the drop-down menu.

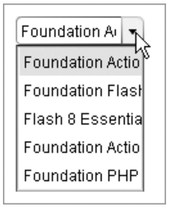


3. Click the plus sign (+) button at the top left of the Values dialog box to enter a label. This will insert a field that reads defaultValue. Click inside this field and type in whatever you want to appear as the first item in the menu. Click the plus sign button to add further items. You can use the minus sign (–) button to remove any that you no longer want, and the up and down arrows to change the position of any item in the menu. For the purposes of this exercise, we've used the names of five friends of ED books, as shown alongside. Click OK when you're done.



	Value
0	Foundation ActionScript for Flash 8
1	Foundation Flash 8
2	Flash 8 Essentials
3	Foundation ActionScript Animation
4	Foundation PHP 5 for Flash

4. Test the movie, and click the arrow to the side of the drop-down menu. You'll see your items listed, but unless you've used very short labels, they're likely to be cut off on the right as shown in the screenshot.

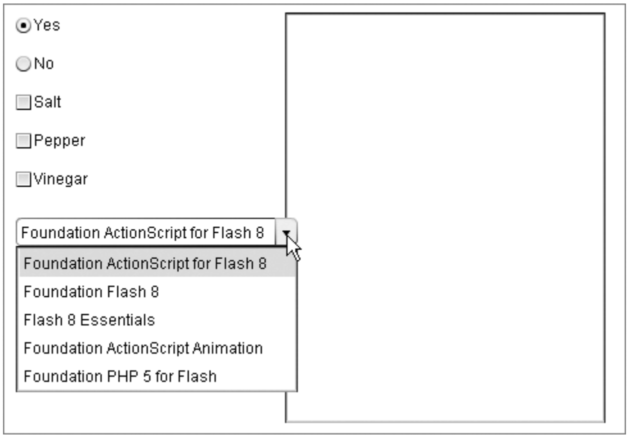


5. To ensure that your items display correctly, you have several options, the first of which is to use the Free Transform tool. But since this is a book about ActionScript, let's skip that. How about `setSize()`? That worked for the `TextArea` component. Open the Actions panel on frame 1 of the actions layer, and add this to the end of the existing script:

```
dropdown_cb.setSize(220);
```

Note that there's only one measurement passed to `setSize()` as an argument—the desired width. ActionScript won't object if you also attempt to set the height of a `ComboBox` component, but the result is far from pretty.

6. Test the movie. As you can see from the screenshot, it definitely works, but it's caused a problem with the `TextArea` component, because the `ComboBox` now overlaps it.

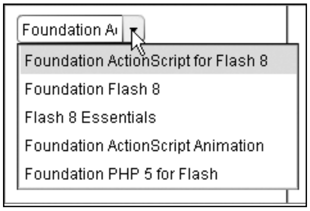

● Yes
○ No
□ Salt
□ Pepper
□ Vinegar
Foundation ActionScript for Flash 8
Foundation ActionScript for Flash 8
Foundation Flash 8
Flash 8 Essentials
Foundation ActionScript Animation
Foundation PHP 5 for Flash

7. In many cases, your best solution will be to adjust your layout so that the two components don't overlap. However, that's not always possible or desirable. ActionScript to the rescue!

Change the last line of script like this:

```
dropdown_cb.dropdownWidth = 220;
```

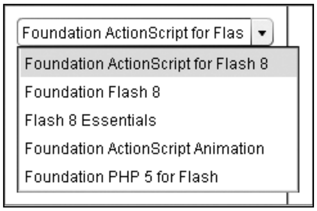
8. Test the movie again. The `ComboBox` component has returned to its original default width, but the contents of the drop-down are now displayed at their correct width.


Foundation A
Foundation ActionScript for Flash 8
Foundation Flash 8
Flash 8 Essentials
Foundation ActionScript Animation
Foundation PHP 5 for Flash

9. It's still not perfect. But there's nothing to stop you from using `setSize()` and `dropdownWidth` in combination like this:

```
dropdown_cb.setSize(200);
dropdown_cb.dropdownWidth = 220;
```

Although the ComboBox component doesn't display the full text in its closed state, it's enough to encourage users to open the menu, and then they can see the full contents. Of course, an even better solution would be to make the first item short enough so that it displays in full.


Foundation ActionScript for Flas
Foundation ActionScript for Flash 8
Foundation Flash 8
Flash 8 Essentials
Foundation ActionScript Animation
Foundation PHP 5 for Flash

10. As we said before, the `label` property of each item is the most important. If you look back at step 3, you'll see that Flash automatically assigns a number to each item in a ComboBox. As with all arrays, the numbering starts at 0. What if you want to assign different values of your own, say for direct interaction with a database? Simple. Set the `data` fields for each item in the same order. You enter the `data` values in exactly the same way as for the `label` property. Just double-click the data field in the Parameters tab of the Property inspector and fill in the Values dialog box. For the purposes of this exercise, the download file uses the ISBN number as the `data` property of each book.

> *Values entered in the* label *field of the* Parameters *tab are automatically treated as strings, because this is the only data type accepted as a* `label` *property. There is no need to enclose them in quotes, but if you do want to display quotes around a label, you need to use a combination of single and double quotes, like this:* '"Foundation ActionScript for Flash 8"'. *The outer pair of quotes will be discarded; only the inner pair will be displayed.*
>
> *Values entered in the* data *field, however, can be of any data type. Flash does its best to guess the most appropriate data type. This can cause unexpected problems if you intend to pass the value of a* data *field to a function or method that expects a particular data type. If you enter the ISBN number for this book as* 1590596188, *Flash will treat it as a number. However, if you enter it with hyphens (*1-59059-618-8*), Flash treats it as a string. Since ISBN numbers sometimes end with an "X", they need to be enclosed in quotes when entered in the* Values *dialog box so they are all treated consistently as strings.*
>
> *Be warned that the* data *and* label *fields in the* Parameters *tab are not linked in any way. They are treated by Flash as two separate arrays. If you change the order of one array, you must—repeat,* must*—change the order of the other one manually.*

11. Populating the ComboBox this way, however, is very restrictive. Say you want to use data imported from an XML document, for instance. Because ActionScript is all about dynamic interaction, you can load your menu items at runtime, saving you the effort of creating a new SWF every time something changes in the menu. Add the following code to the end of the script in the Actions panel:

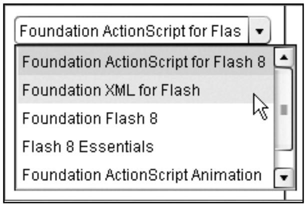
```
dropdown_cb.addItem("Foundation XML for Flash", "1590595432");
```

The addItem() method adds a new item to the end of the existing menu in a ComboBox. The first argument is the label, and the second is the data property. The second argument is optional.

12. If you test the movie now, you'll see the new item at the bottom of the menu. Next, change the code in the previous step like this:

```
dropdown_cb.addItemAt(1, "Foundation XML for Flash", "1590595432");
```

The code looks almost identical, but this time it uses the addItemAt() method instead of addItem(). It also takes an extra argument—a number, which comes before the label property. The number indicates the position at which you want the new item to be inserted into the menu. Because arrays start at 0, this code will insert Foundation XML for Flash as the second item in the menu and automatically move the others down one position. Test it and see.


Foundation ActionScript for Flas
Foundation ActionScript for Flash 8
Foundation XML for Flash
Foundation Flash 8
Flash 8 Essentials
Foundation ActionScript Animation

13. Now that you have six items in the menu, they don't all display without the need to scroll downward. The Parameters tab of the Property inspector lets you change the value of rowCount, which defaults to 5, but you can do it with ActionScript, too. Add the following line at the end of your existing script:

```
dropdown_cb.rowCount = 6;
```

Test the movie again. When you open the drop-down menu, all six items will display without the need for scrolling.

Extracting data from a ComboBox

Continue working with the same FLA. The final code for this exercise is in components04.fla.

1. Open the Actions panel on frame 1 of the actions layer and add the following code to the end of the script:

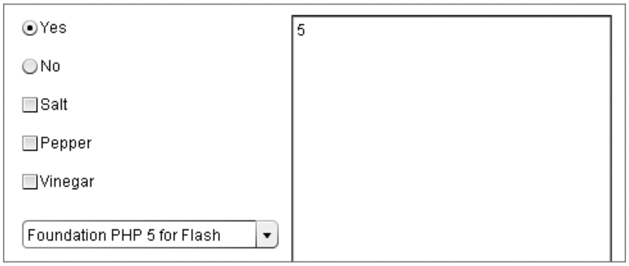
```
function queryCombo(evtObj:Object):Void {
  output_ta.text += evtObj.target.selectedIndex + "\n";
}
var comboListener:Object = new Object();
comboListener.change = queryCombo;
dropdown_cb.addEventListener("change", comboListener);
```

This sets up a function called queryCombo() to extract information from a ComboBox component, together with a listener object that's designed to respond to the change event of dropdown_cb. The function uses the event object to identify the component as evtObj.target. You could rewrite the function without the event object like this:

```
function queryCombo():Void {
  output_ta.text += dropdown_cb.selectedIndex + "\n";
}
```

Both versions of the function work identically. The advantage of using the event object, however, is that it will work with *any* ComboBox component. The second version works only with dropdown_cb. So, although using the event object may feel less intuitive, it's much more efficient in the long run.

2. Test the movie and select an item from the drop-down menu. You should see something like the following screenshot.



◉ Yes

○ No

□ Salt

□ Pepper

□ Vinegar

Foundation PHP 5 for Flash ▾

5

If you're familiar with JavaScript, you'll recognize that selectedIndex gives you the position of a selected item in the array of a drop-down menu. *Foundation PHP 5 for Flash* (you can guess who wrote this section, can't you?) is currently the sixth item in the menu, so 5 appears in the text area (0, of course, being the index of the first item).

3. Open the drop-down menu again and select the *same* item as you selected the first time. Nothing happens. The reason should be obvious, but it is sometimes difficult to grasp. The event is called change, so if nothing changes, the event doesn't fire.

4. Very often, change is exactly what you want, because you don't need to update anything if there's no change. However, there are occasions when you need to register what's been selected every time a menu is accessed, even if there is no change. Amend the final two lines of your script like this:

```
comboListener.close = queryCombo;
dropdown_cb.addEventListener("close", comboListener);
```

The ComboBox component now responds to the close event, which is triggered every time the menu is closed (makes sense, really). If you test the movie now and select the same item twice in succession, its index number should appear in the text area each time.

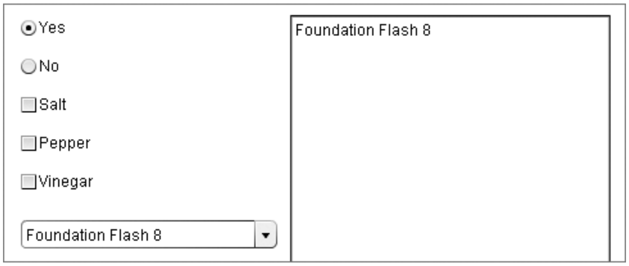
5. Using selectedIndex to get the array index has its uses, but it's not always the most helpful. How about getting the value of the label property? Change the code inside the queryCombo() function like this:

```
output_ta.text += evtObj.target.selectedIndex.label + "\n";
```

6. Test the movie. Whoa! It doesn't work. Instead of the name of a book, you get undefined. That's because selectedIndex returns only the array index. To get access to the label property, you need to use selectedItem instead, like this:

```
output_ta.text += evtObj.target.selectedItem.label + "\n";
```

7. Test it again. That's better.



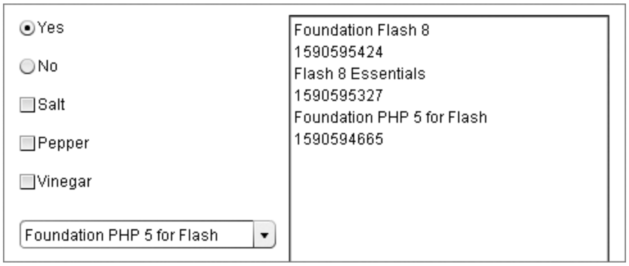
● Yes
○ No
□ Salt
□ Pepper
□ Vinegar
Foundation Flash 8 ▾

```
Foundation Flash 8
```

8. You can also retrieve the data property of a menu item using selectedItem. Amend the queryCombo() function like this:

```
function queryCombo(evtObj:Object):Void {
  output_ta.text += evtObj.target.selectedItem.label + "\n";
  output_ta.text += evtObj.target.selectedItem.data + "\n";
}
```

If you select several items from the menu, you will now see both the book's title and its ISBN displayed in the text area. Extracting the data value is particularly useful for working with a database. Even if you don't have a database, you could store the filenames of movie clips or photos that you want to load into your main movie when selected from the menu.



● Yes
○ No
□ Salt
□ Pepper
□ Vinegar
Foundation PHP 5 for Flash ▾

```
Foundation Flash 8
1590595424
Flash 8 Essentials
1590595327
Foundation PHP 5 for Flash
1590594665
```

> *When working with a* `ComboBox` *component, remember that* `selectedIndex` *returns the array index of an item in the menu. To get access to the* `label` *and* `data` *properties, you need to use* `selectedItem`.

Using code hints with components

Because components aren't part of core ActionScript, the Actions panel doesn't automatically pop up handy code hints in the same way as it does for movie clips, strings, and arrays. However, you can turn them on. Unfortunately, it's not an option that you can set once and then forget; you need to do it for every FLA individually. Still, if you're not good at remembering the commands or have problems spelling them correctly (and with the correct mixture of lowercase and uppercase), code hints can be useful, especially in a long script. Here's how you turn them on.

Turning on code hints for a ComboBox component

1. Open a new Flash document and drag an instance of the ComboBox component onto the stage. In the Property inspector, give the component an instance name of myMenu_cb.
2. Insert a new layer called actions and open the Actions panel on frame 1 of the new layer.
3. Type the following command at the top of the Actions panel:

   ```
   import mx.controls.ComboBox;
   ```

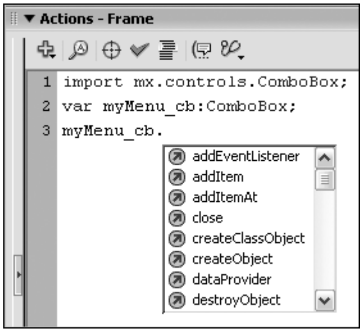
   This tells Flash where to find the ComboBox class. You've already met import in Chapter 9, when working with the Tween class. It doesn't physically import anything into your document, but it's used by Flash to access code that isn't part of core ActionScript.
4. On the next line, define a variable using exactly the same name as the instance name you assigned the ComboBox component in the Property inspector, and set its data type to ComboBox, like this:

   ```
   var myMenu_cb:ComboBox;
   ```

5. To start adding items to the combo box with ActionScript, you need to use the addItem() method, so type this on the next line:

   ```
   myMenu_cb.
   ```

   As soon as you type the period after the instance name, code hints specific to the ComboBox component will pop up as shown alongside.



```
import mx.controls.ComboBox;
var myMenu_cb:ComboBox;
myMenu_cb.
```

To get code hints for other components, you need to add the appropriate import command at the top of your script. The command for all User Interface components takes the following form:

```
import mx.controls.componentClassName;
```

You have probably noticed that, in all the exercises, we have given components instance names that end with `_rb` *for* `RadioButton`, `_ch` *for* `CheckBox`, `_cb` *for* `ComboBox`, *and* `_ta` *for* `TextArea`. *This is nothing more than a convention, which serves as a reminder of the type of component being used. Unlike* `_mc` *or* `_txt`, *these suffixes don't trigger code hints on their own. You need to turn them on specifically by importing the relevant class.*

With ActionScript 2.0 strict typing becoming more common, some developers are abandoning the use of `_mc` *and* `_txt`, *too. As you gain more experience, you will develop your own style. However, following established conventions will make your code easier to understand and maintain. But doesn't it also make code easier to steal? Maybe it does, but if you're writing code so sophisticated that others might be tempted to steal it, you're probably writing material that such people probably couldn't understand anyway.*

Loading components at runtime

Because version 2 components are precompiled movie clips, you're not limited to using them in the Flash authoring environment. You can also load and unload them dynamically at runtime. There are two ways to do this: the official way and the unofficial way. Let's start with the official one.

Using createClassObject() to load a component

1. Open a new Flash document and drag a `TextArea` component from the Components panel directly into the Library panel. If you just want to look at the finished code, it's in `runtime01.fla`.
2. Open the Actions panel and insert the following code:

```
import mx.controls.TextArea;
this.createClassObject(TextArea, "myText_ta", 0);
myText_ta.setSize(250, 150);
myText_ta.move(40, 30);
myText_ta.text = "Hi, I'm a dynamically generated TextArea component!";
```

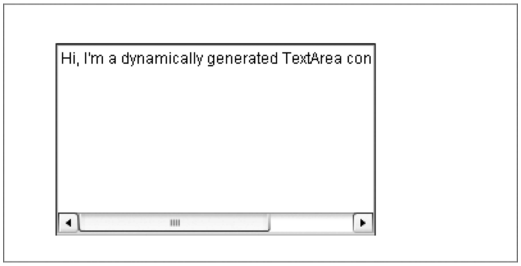
The first line imports the `TextArea` component class, which enables you to use the `createClassObject()` method with the `TextArea` component in your Library.

The `createClassObject()` method is applied to the current timeline, identified by `this`, and takes three arguments: the class name of the component (`TextArea`), a string that will be used as the component's instance name, and the depth you want to load it into.

The remaining three lines change the size of the `TextArea` component to 250 × 150 pixels, position it at x-coordinate 40 and y-coordinate 30, and set some text. Note that the method used to position components is `move()`. It's part of the `UIObject` class that's common to all components.

> *Although components have* x *and* y *properties, they are read-only. You cannot change the position of a component by attempting to assign a new value directly to either* x *or* y*. Another point to note is that they're called just plain* x *and* y*, not* _x *and* _y*. Underscores are out with component property names. The* width *and* height *properties are similarly read-only. To change size and position, use the* setSize() *and* move() *methods, respectively.*

3. Test the movie. You should see something similar to the following screenshot.


Hi, I'm a dynamically generated TextArea con

4. Oops! Those automatic scrollbars are handy, but this time the component has generated a horizontal one! This is a nasty trick that the TextArea component plays on you. When you place one directly on the stage, the wordWrap property is set to true. However, when you load a TextArea component at runtime, wordWrap is set to false. Add the following line to the code immediately before the last one:

```
myText_ta.wordWrap = true;
```

When you test the movie again, the scrollbar should have vanished and the text will wrap normally inside the component. Keep the FLA open to use in the next exercise.

You can use createClassObject() with all the User Interface components. Just substitute the correct class name in the import command and as the first argument to createClassObject().

Using createObject() to load a component

1. In the FLA that you created in the previous exercise, amend the ActionScript so that it looks like this (or look at the code in runtime02.fla):

```
this.createObject("TextArea", "myText_ta", 0);
myText_ta.setSize(250, 150);
myText_ta.move(40, 30);
myText_ta.wordWrap = true;
myText_ta.text = "Hi, I'm a dynamically generated TextArea component!";
```

The difference is that the `import` command in the first line has been deleted. What has now become the first line uses `createObject()` instead of `createClassObject()`, and the first argument is enclosed in quotes.

2. Test the movie. You'll see that it compiles much quicker and that the result is identical.

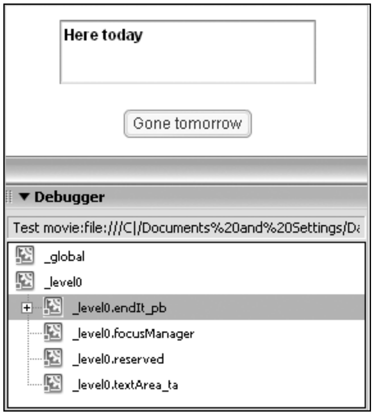
Since `createObject()` is faster and involves less coding, it's a mystery as to why `createClassObject()` is the official way of doing this. Sometimes unofficial methods are risky because they're undocumented. That's not the case with `createObject()`. It's part of the `UIObject` class, but the documentation says it's "generally used only by component developers and advanced developers." Welcome to the advanced ranks!

For a more complex example of components being loaded or manipulated at runtime, take a look at `form.fla` *in the download files for this chapter. Give yourself an amateur personality test while you're at it.*

Removing components

Suppose a component that you have loaded at runtime is no longer needed. Rather than just setting its `visible` property (again, no underscore) to `false`, it's better to get rid of it to free up memory and other resources. You do this with the `destroyObject()` method.

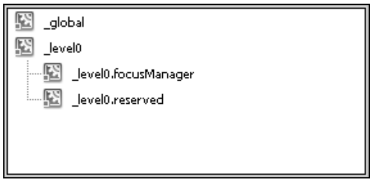
There's an example of the `destroyObject()` method in action in `remove01.fla` in the download files for this chapter. If you test the movie in debug mode, you'll see there are two components, a `TextArea` with an instance name of `textArea_ta` and a `Button` with an instance name of `endIt_pb`, as shown in the screenshot alongside.



Clicking the Gone tomorrow button triggers an event that runs a function called `vanish()`, which looks like this:

```
function vanish():Void {
  destroyObject("textArea_ta");
  destroyObject("endIt_pb");
}
```

As soon as you click the button, both components don't just disappear from view, but are completely removed from the movie, as the Debugger panel confirms.



Removing listeners

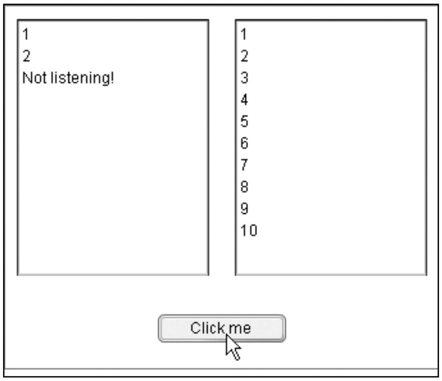
Sometimes you don't want to remove a component, but you want to stop some events from being handled. To do this, simply unregister the listener from the broadcast using removeEventListener(). The syntax looks like this:

```
target.removeEventListener("event", listenerObject);
```

There's an example of removeEventListener() in remove02.fla in the download files. The key part of the script is the threeTimes() function, which looks like this:

```
function threeTimes():Void {
  if (count == 3) {
    textArea_ta.text += "Not listening!";
    clickMe_pb.removeEventListener("click", buttonListener);
  } else {
    textArea_ta.text += count++ +"\n";
  }
}
```

Once the timeline variable count gets to 3, the fourth line of the threeTimes() function removes the event listener buttonListener from clickMe_pb and stops displaying the value of count in a text area. However, the removal of buttonListener doesn't affect a second listener, which continues responding, as shown alongside.



```
1
2
Not listening!
```

```
1
2
3
4
5
6
7
8
9
10
```

Study the code in the download file. The important thing to realize is that it's not threeTimes() checking the value of count every time the button is clicked and deciding not to display the value. It's *not listening* to the button any more. With a trivial example like this, the difference in the use of resources is infinitesimal. In a more complex application, conservation of resources becomes far more important.

Now that you're now moving into the more advanced stages of ActionScript, attention to this sort of detail is what will set you apart from a beginner. ActionScript is no longer just for creating simple animation effects onscreen, however cool they may be. It's evolving into a highly sophisticated programming language. Although you don't have to go to the most advanced levels if you don't want to, full-fledged object-oriented programming (OOP) in ActionScript is now a reality.

ActionScript and OOP

ActionScript underwent a major change in September 2003 with the release of ActionScript 2.0, and it's likely to undergo another, perhaps even more significant change during the lifetime of this book. Detailed plans have been announced for ActionScript 3.0, which is expected to make its appearance sometime during 2006 in a program called Flex 2. In many ways, Flex is similar to Flash, as both products were originally developed by Macromedia and are now part of the Adobe family. Although it's difficult to predict future developments, it looks as though Flex will concentrate on interactive forms using Flash components, whereas Flash will maintain its traditional focus on the designer end of the market. The plan, however, is for both Flex and Flash to adopt ActionScript 3.0.

The future road map for ActionScript

You're probably groaning now that you've wasted your time struggling through this book when everything is about to change. Far from it. First of all, Macromedia/Adobe is on record as saying that future versions of Flash Player will maintain backward compatibility with both ActionScript 1.0 and ActionScript 2.0. So, everything you have learned in this book will continue to work in the years to come. Even better, we've deliberately written this book to make the transition to ActionScript 3.0 easier. For instance, using data typing is entirely optional in ActionScript 2.0, but in ActionScript 3.0 missing data types will generate a warning. So we've encouraged you to use typing throughout the book. (You have been using it, haven't you?) We've also removed details of techniques that we know you'll no longer be able to use. The most important of these is the use of prototypes.

> ***Prototypes*** *are the ActionScript 1.0 way of adding new methods and properties to built-in classes, such as* `MovieClip`*. You can still use them in ActionScript 2.0, but their use is frowned upon and can cause unexpected problems at compile time. They have been completely removed from ActionScript 3.0. We don't cover prototypes in this book, and you should be wary of any online articles or books that recommend using* `MovieClip.prototype`*. Although an important and useful technique in the past, it's now completely out of date.*

ActionScript is being brought more and more into line with an internationally recognized coding standard defined by an organization known as Ecma International (`www.ecma-international.org`), which is also responsible for JavaScript and C# standards, among others. According to Macromedia/Adobe, the changes are needed because developers have pushed the ActionScript Virtual Machine (AVM1) inside Flash Player to its limits, and AVM2, which will be used by ActionScript 3.0, executes up to ten times faster than is currently possible. The price—in terms of effort—that developers will have to pay is adherence to much stricter standards. ActionScript 2.0 was the first step toward giving the language the characteristics of a genuine OOP language. ActionScript 3.0 will take that process much further.

The ActionScript that you have learned is already class-based, so the fundamental principles of working with classes, methods, and properties will remain unchanged. But the components that you worked with in the first part of this chapter give a strong indication of ActionScript's future direction. Instead of being able to set the width and height of a component directly by assigning new values to `width` and `height`, you must use the `setSize()` method. The `width` and `height` properties are read-only. This follows an OOP principle called **encapsulation**, which is a highfalutin way of saying black box. The idea is that objects created in your scripts should be perfect black boxes, and the properties of an object cannot be changed arbitrarily. In OOP, you can limit access to certain properties and methods by declaring them `private`—in other words, only internally accessible within the class itself.

If all of this is beginning to sound like Vogon poetry, don't worry. Although you will need to make some changes to the way you write ActionScript to take advantage of the speed improvements promised by ActionScript 3.0, you don't need to concern yourself with the details of OOP—unless of course you want to. ActionScript is already based on objects, methods, and properties, and it will continue to be so. What's important about ActionScript 2.0 and 3.0 is that they give you the freedom to create your own classes in a fully OOP way.

Is OOP for me?

Although you have been using objects, such as movie clips and arrays, right from the beginning of this book, everything you have done so far is called **procedural programming**. You create a movie clip, give it an instance name, build a few functions, and the finished movie does what you want (at least, you hope it does). You then move on to the next project and do it all over again. Of course, you may copy and paste some useful functions into the new movie, but all the time your thinking is driven by "what happens next."

Object-oriented programming takes a different approach. Say you're building a space invaders game. Instead of using a generic movie clip for your aliens, you use an `Alien` object, which automatically has built into it all the diving, swooping, and attacking characteristics that it needs. ActionScript doesn't have an `Alien` class, so you have to design it yourself. Once you have designed it, though, it can be made available in all your projects. If you're working in a team, other members don't even need to know how the `Alien` class works internally. If they have created an `Alien` instance called `darkShip`, all they need to do is type darkShip.attack() and it will happen.

As you will see shortly, creating your own classes is very easy. The reason we've put them in a chapter titled "Advanced ActionScript" is not because the code is difficult to write or understand, but because deciding *what* to turn into a custom class and designing your class structure is the hard part. Let's take a look at some basic questions:

- What new features will OOP coding styles give me?
- Will OOP help me write code quicker or more efficiently?

Contrary to popular opinion in some quarters, writing object-oriented code will give you *no* new features to play around with. All it gives you is a more structured development environment.

In terms of speed, using OOP coding styles will actually *slow you down* for small projects, unless you take the time to get proficient in it. Object-oriented code requires well-defined interfaces in your code, and this can also make your code longer.

Um, we feel a third question coming up:

- Why the heck do I need to learn OOP, then?

In many cases, you don't need OOP. Small projects can be created using the styleless but structured code you've been using throughout the book so far. If your code won't be reused in anything else (i.e., you'll start your next web project from scratch), then OOP isn't appropriate.

Here are the instances when OOP *is* appropriate:

- **When you're building large applications**: Writing the first thing that comes into your head is a good way of getting a small, 30-line ActionScript trick-shot working, but for anything bigger, you need a more structured approach. Something like the Futuremedia site is about as big as you can get with a general modular coding style that doesn't use OOP before you need to rethink your strategy.

- **When you want to reuse your code**: If you're going to use the same code often, it makes a lot of sense to make that code a class or part of an existing class. When you do this, the interfaces to your code become methods and properties, both of which are totally recognizable to you *and* any other ActionScript coder. This makes your code easy to use in new Flash movies—you can simply treat the new features provided by the code as an add-on to ActionScript. Components are class based, and this makes them easy to add to your code.

You may have assumed that the methods and properties that you used to access the `TextArea` *and other components are part of ActionScript. They aren't; instead, they're the interfaces to the code that drives the components. They're a good example of how closely integrated a class-based add-on can be. It can begin to look like you're using a new part of ActionScript itself, rather than code written by someone else!*

So, in general, you have increased flexibility and structure versus the increased workload needed to set all this extra stuff up.

Another point worth mentioning now is that using class-based code doesn't make the final code any more efficient. In fact, Flash will convert a lot of your class-based code into simpler modular code as part of the conversion from FLA to SWF.

The only advantage of using one style over another relates to managing the coding project. Choose the one that best fits what you're comfortable with and the size of the task.

- Well-written modular code is good for most website design.
- Class-based code is useful when you want to build major additions to ActionScript (such as components) or you want to build complex Flash sites.
- Some problems involve working with classes, so it makes sense to use classes in your solution. This works well when you're handling large amounts of information (such as databases) or you're working with fairly complex transmission protocols as part of your design. By integrating the class definitions (of the objects you're interfacing with) into your solution, you're better positioned to manage the interfaces.

It has to be stressed that there's no pecking order here. Suggesting that class-based code is better than procedural code is much the same as saying that a hammer is better than a screwdriver. The statement is nonsense unless you also include *the task*. A hammer is good for hitting nails, but if you always think in terms of a hammer, all your solutions will involve hitting things, which may not always be appropriate.

If you plan on taking advantage of ActionScript 3.0 when it becomes available in Flash, you'll want to learn as much about coding with classes as you can. At the time of this writing, ActionScript 3.0 is available as a public beta from `http://labs.macromedia.com/technologies/actionscript3`*, where you can also find the extensive draft documentation.*

How class-based coding works

Flash class-based programming involves the use of one or more classes that *extend* your code toward the solution. For example, if you're creating a general set of scripts to program games, you might need the following:

- A general movement class, `Mover`.
- A more aware class, `Navigate`. As well as moving the movie clip sprites, this class would know exactly where the sprite can and can't move, and take appropriate action.
- A final class, `Animate`. As well as moving and navigating, this class would control the appearance of the sprite, so an alien that has just been hit will explode, for instance.

The way to write class-based code would be to start with the simplest class, `Mover`. The `Mover` class would contain methods that allow movement to specified points on the stage.

The next class, `Navigate`, would extend `Mover` by preventing any movement that stops the movie clip, for example, by preventing it from moving offscreen.

Finally, the third class, `Animate`, might be designed so that more advanced methods are available. You might have additional methods that can ensure a movie clip always points in the direction it's traveling.

The cool thing about this is that you have different levels of problem solving.

- The low-level classes such as `Mover` are very general and would be used in *all* games.
- Classes like `Navigate` would be used in most games.
- Classes like `Animate` would be used in games in which you actually wanted sprites to point in the direction they're moving.

So, in the class-based approach, you end up with different *layers* of problem solving. You start at the foundation, building the simplest classes, and then you use these classes to build more complex classes.

The advantage of this is that not only do you have structured code, but you also have a structured way of splitting complex problems in a series of levels of problem solving, rather like building a house:

1. Build the foundation (the basic or core classes).
2. Extend the foundation by adding more functionality.
3. Keep repeating the last step, building new classes until you've solved your problem.

Let's try this out with a simple example. We recommend you try this using Flash Professional 8, although you'll be able to get by using Flash Basic 8. ActionScript classes *must* be defined in an external file. The Professional version has a dedicated screen for writing and editing class files, but the Basic version doesn't. The following exercise gives instructions for both versions, but users of Flash Basic 8 won't be able to check the syntax or tidy up the script with the Auto format button.

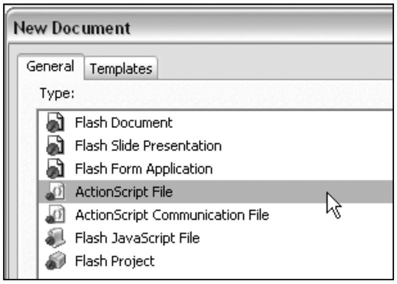
Many Flash developers—even those with the Professional version—prefer to use a stand-alone editor for ActionScript. Sham's favorite is SciTE|Flash from `www.bomberstudios.com/sciteflash`*. The program hasn't been updated for some time, so you also need to get the ActionScript 2.0 configuration files from* `http://mizubitchy.antville.org/stories/519068`*. A big advantage of SciTE|Flash is the ability to collapse sections of code when you're not working on them—a feature sorely missing in Flash 8. Unfortunately, SciTE|Flash isn't available for Mac OS X. An alternative for both Windows and Mac is SE|PY. To get it, go to* `www.sepy.it` *and follow the download link for the latest stable version. The* `.dmg` *file is for Mac, and the* `.exe` *file is for Windows. Both SciTE|Flash and SE|PY are free.*

Creating and using the low-level (base) class

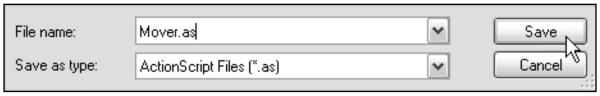
The lowest level class is called `Mover`, which will be used to give movie clips the same inertial movement that you experimented with in Chapters 9 and 10. If you don't want to type in the script yourself, the finished version is in `Mover.as` in the download files for this chapter.

1. If you are using Flash Professional 8, make sure the Actions panel is minimized, then choose File ➤ New, select ActionScript File from the General tab, and click OK.

   You won't have this option in Flash Basic 8, so just create a new Flash document and open the Actions panel.



2. The scripting panel that opens in Flash Professional 8 is just like a large version of the Actions panel that covers most of the screen. All other main panels are grayed out. Choose File ➤ Save and navigate to the folder where you are building the test files for this book. The Save As dialog box automatically sets the Save as type option to ActionScript Files (*.as). Name the file Mover.as (with an initial capital M), and click Save.


File name: Mover.as

Save as type: ActionScript Files (*.as)

   You can't do this in Flash Basic 8, so you need to save the complete FLA as an ordinary Flash document. When the script is finished, you can then export it to an AS file.

3. Whichever version of Flash you're using, type the following script into the document you have just created:

```
class Mover {
  //
  // PROPERTIES
  //
  // CONSTRUCTOR
  public function Mover() {
    // initialize the new Mover object
  }
  //
  // PUBLIC METHODS
  //
  // PRIVATE METHODS
}
```

What you have defined here is just the general structure of the class, but all classes follow the same pattern. The first thing to note is that you begin the definition using the class keyword followed by the name of the class, which must be *exactly* the same as the file it's saved in (apart from the .as filename extension).

Remember that class names normally begin with an uppercase letter. Get into the habit of naming them so.

Within the base class definition, you define properties and methods. You also need a constructor function to create new instances of your class. At the moment, the constructor function doesn't contain any code, but you've probably already noticed that it has exactly the same name as the class. So that's three things that all have the same name: the AS file that the class is stored in, the class itself, and the constructor function. But what's that public doing in front of the function keyword?

Both properties and methods can be one of two things: private or public.

- A **public** part of the class definition is available to ActionScript outside the class file. A public variable becomes an instance *property* when using the class back in the world of the FLA, just like the _x property of a movie clip is a freely accessible (public) property, and all movie clips have one.
- A **private** part of the class definition is *not available* to ActionScript outside the class file. A private property is used internally by the class, and it isn't for use by the main application.

An easy way to understand the difference between public and private is to think about driving a car. Most drivers know very little about the inner workings of their car's engine. They just use the ignition, accelerator, brake, and steering wheel. These are the public parts of a car. Everything inside the engine is private. Even if you're happy tinkering about with the engine, you do so only when the car isn't moving. The public parts of a class are the bits that everyone can use. By keeping private methods and properties locked up in a (metaphorical) black box, your code is more robust, because it prevents inexperienced people from making changes that could upset the performance of your object.

4. The Mover class is going to implement the inertial movement from Chapter 9. It's defined in two parts: first the property definitions and then the methods. The first section defines the properties of Mover. Add the following code under the PROPERTIES comment:

```
// PROPERTIES
private var theTarget:MovieClip;
private var theClip:MovieClip;
private var distX:Number;
private var distY:Number;
```

As you can see, all of the properties are private, so they'll be used only internally within the class definition.

- theTarget: This is the name of the movie clip you're moving.
- theClip: This is the name of a dummy movie clip that will be used to control the movement with an onEnterFrame event handler. You need to do this with a separate clip in case the movie clip that you're moving already has an onEnterFrame event handler of its own.
- distX and distY: These two internal values are used to work out the inertia motion. Again, the main FLA isn't required to know about these values, so you hide them.

5. Next up, you have the constructor function. This function is called whenever the main code using this class creates a new instance with the new keyword. Amend the code as shown here:

```
// CONSTRUCTOR
public function Mover(targetClip:MovieClip) {
  theTarget = targetClip;
}
```

The constructor accepts a movie clip as an argument and assigns it to the private variable theTarget. Constructor functions are frequently very simple, like this. They just create the instance and set default properties, if any.

> *There are two important things to note about the constructor function: it isn't typed, and it doesn't return anything. If you attempt to return a value from a constructor function, it will generate an error. Even though constructors don't return a value, you shouldn't use* :Void *after the parentheses in the function declaration, either.*
>
> *A class can have only one constructor function, and in almost all cases, it needs to be* public. *Otherwise, it would be impossible to create new instances. It is, however, possible to declare a constructor as* private *and then use a* public *method to ensure that only one instance of a class is ever created. This is a design pattern known as a* Singleton. *Design patterns are covered in* Object-Oriented ActionScript for Flash 8 *(friends of ED, ISBN: 1-59059-619-6).*

6. After the constructor, you need to define the methods that the class will use. A method is simply a function and is declared with the function keyword. There are two public functions, moveInertia() and getTarget(). The first of these functions creates the dummy movie clip, sets the data that the dummy movie clip needs to know, and attaches the onEnterFrame script to the dummy clip. Add the following script to your class file:

```
// PUBLIC METHODS
public function moveInertia(x:Number, y:Number):Void {
  theClip = theTarget.createEmptyMovieClip("moveInertiaHolder", ➥
theTarget.getNextHighestDepth());
  theClip._x = x;
  theClip._y = y;
  theClip.onEnterFrame = moveClip;
}
```

The moveInertia() function will be accessible in ActionScript, so it's declared as public. It takes two arguments: the x- and y-coordinates of where the target movie clip is to end up. The dummy clip is moved to those coordinates. The onEnterFrame script, moveClip(), performs the actual movement. It's a private function that will be defined in a moment.

7. The second public function simply returns the name of the target movie clip that you're controlling with this class. Add the following script immediately below the code you entered in the previous step:

```
public function getTarget():MovieClip {
  return theTarget;
}
```

The value returned is a movie clip, so you need to set MovieClip as the method's return type.

8. Finally, the function that actually executes the move is moveClip(). It's used by moveInertia(), and nothing else needs to look at this code, so you make it inaccessible outside the class by making it private. Inside this function, you have your old friend from Chapter 9, the inertia equation. You stop the animation when the target clip is within 1 pixel of its target position with the if statements at the end of the function, and you do this by deleting the dummy clip. Insert the following code in your class definition:

```
  // PRIVATE METHODS
  private function moveClip():Void {
    var me:Object = this;
    var mc:MovieClip = me._parent;
    distX = me._x - mc._x;
    distY = me._y - mc._y;
    mc._x += distX/4;
    mc._y += distY/4;
    if (Math.abs(mc._x - me._x)<1) {
      if (Math.abs(mc._y - me._y)<1) {
        mc._x = me._x;
        mc._y = me._y;
        me.removeMovieClip();
      }
    }
  }
}
```

Inside this function, me refers to the dummy movie clip to which moveClip() is assigned as the onEnterFrame event handler, and mc refers to the movie clip that's being moved. The correct data type for me should be Mover, but this causes type mismatches at compile time because Mover doesn't have _x or _y properties. There is a very simple solution, which we'll show you in a moment, but use Object as the data type for the time being.

9. If you're using Flash Professional 8, click the Check Syntax icon at the top of the script pane to make sure there aren't any errors, and then save Mover.as.

If you're using Flash Basic 8, you won't be able to check the syntax, so just click the icon at the top-right of the Actions panel and choose Export Script from the menu that appears. Navigate to the folder where you are creating the test files for this book, and save the script as Mover.as (remember that it begins with an uppercase M).

10. Open a new FLA and save it in the same folder as Mover.as. Create a small square or circle in the center of the stage. Convert it to a movie clip, and give it an instance name of myClip.

11. Add a new layer called actions, and enter the following code in the Actions panel on frame 1 of the actions layer:

```
var myMover:Mover = new Mover(myClip);
myMover.moveInertia(10, 10);
```

This creates a new instance of the Mover class called myMover, so the correct type becomes Mover, not MovieClip. You can now control the movement of myClip as myMover, using the moveInertia() method of your new class. If you test the movie, the square or circle should move from the center of the stage to the top-left corner (coordinates 10, 10) with inertia movement.

This won't work if you haven't saved the FLA, because the compiler needs to be able to find Mover.as. *If it still doesn't work, check that the FLA and* Mover.as *are both in the same folder, and that you have spelled the filename, class name, and constructor function name consistently with the same mixture of uppercase and lowercase. If all else fails, try* Mover.as *and* testClass01.fla *in the download files.*

12. Now try adding this line to the script:

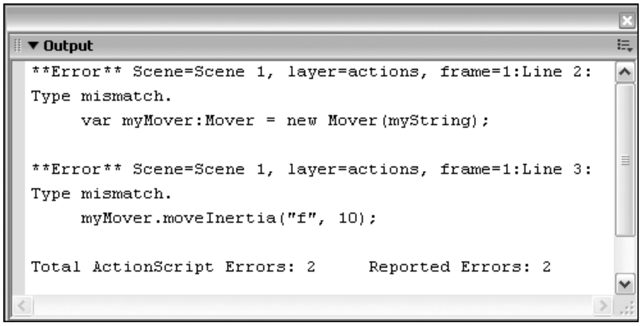
```
trace(myMover.getTarget());
```

This uses the getTarget() method to remind you which movie clip you're controlling. Although you wouldn't use trace() in a real application, it's normal to create this sort of method in a class to extract information about an object. For instance, you may want to create a class that draws particular shapes automatically. By making the height and width private properties, you prevent accidental changes to the size of your shapes. However, you still need a way to find out how big a shape is. Since private properties are hidden, if you create a get*PropertyName*() method like this, you can return the value of the property without compromising its integrity.

13. Finally, try this:

```
var myString:String = "Move me baby";
var myMover:Mover = new Mover(myString);
myMover.moveInertia("f", 10);
```

This code has two errors. You pass a string to the class when creating your instance in line 2, and you pass the string f instead of a number in the last line. You'll find that meaningful error messages come back. Flash tells you that you're sending the wrong type of data in both cases. So even with your own classes, Flash performs type checking.



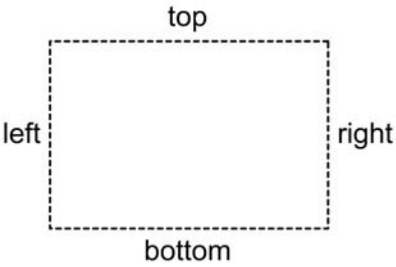
```
**Error** Scene=Scene 1, layer=actions, frame=1:Line 2:
Type mismatch.
     var myMover:Mover = new Mover(myString);

**Error** Scene=Scene 1, layer=actions, frame=1:Line 3:
Type mismatch.
     myMover.moveInertia("f", 10);

Total ActionScript Errors: 2     Reported Errors: 2
```

Extending classes

The structure of OOP allows you to easily *extend* your classes, creating a more advanced class over an existing one, rather like building a high-tech and modern house starts with a simple concrete foundation.

> *When you extend a class, the new class is called the **subclass**, and the class that has been extended is the **superclass**.*

As a simple example, you'll create a new class, Navigator, which is the beginnings of a more complex set of motion routines based on the more primitive (**base class** or **lower-level class**) Mover class. Navigator allows you to limit the inertia movement within a bounding box. At the moment, you can use the Mover class to make your target clip go anywhere.



The Navigator class extends the current method Mover.moveInertia() by never moving the target movie clip outside the bounding box defined by the four dimensions shown.

Since the code is fairly short, we'll present the entire listing first and then explain the important features highlighted in bold. You can find the script in Navigator.as in the download files for this chapter.

```
class Navigator extends Mover {
  //
  // PROPERTIES
  private var left:Number;
  private var right:Number;
  private var top:Number;
  private var bottom:Number;
  private var x:Number;
  private var y:Number;
  //
  // CONSTRUCTOR
  public function Navigator(target:MovieClip, l:Number, r:Number, ➥
t:Number, b:Number) {
    super(target);
    left = l;
    right = r;
    top = t;
    bottom = b;
  }
  // METHODS
  public function moveInertia(xPos:Number, yPos:Number) {
    x = xPos;
    y = yPos;
    if (xPos < left) {
      x = left;
    } else if (xPos > right) {
      x = right;
    }
    if (yPos < top) {
      y = top;
    } else if (yPos > bottom) {
      y = bottom;
    }
    super.moveInertia(x, y);
  }
}
```

To extend a class, you use the keyword extends followed by the name of the base class, like this:

```
class Navigator extends Mover
```

When you extend a class, the subclass automatically inherits all the properties and methods of the superclass, so there's no need to declare them again. However, you can also **override** existing properties or methods, as we have done here with moveInertia(). The benefit of overriding a method is that you keep the same method name, but can get it to act in a different way. Before looking at moveInertia(), though, we need to take a look at the constructor function.

The first line inside the constructor looks like this:

```
super(target);
```

This uses the super() operator to pass target to the superclass Mover. In essence, it's telling Flash "Start with Mover, and add my new methods and properties to that class (to create a more advanced class)." Without this, Navigator would not be able to inherit methods and properties originally belonging to Mover.

The function moveInertia() has a set of if statements that check the target position (x, y) the user specifies for the animation. If the target position is outside the bounding box, it's readjusted to keep it inside. The last line of the function calls the method moveInertia() from the superclass. Because both Mover and Navigator have methods with the same name, you need to use the super keyword again to tell Flash that it's the version from the superclass that you want to use, like this:

```
super.moveInertia(x, y);
```

Testing the Navigator class

1. Open the FLA that you used in the previous exercise to test the Mover class, and change the code in the Actions panel like this (or use testClass02.fla):

   ```
   var myNavigator:Navigator = new Navigator(myClip, 100, 360, 80, 380);
   myNavigator.moveInertia(10,10);
   ```

   This sets the left and top boundaries to 100 and 80 pixels, respectively, but the target is still 10,10.
2. Make sure that Navigator.as is in the same folder, and test the movie. The shape now comes to rest much farther away from the corner than before.
3. Now add this to the code and test the movie again:

   ```
   trace(myNavigator.getTarget());
   ```

   Even though the Navigator class doesn't have a getTarget() method of its own, it inherits it from the Mover class and displays the name of the movie clip that you have just moved.

These are simple examples of how you build and deploy custom classes in ActionScript 2.0. In a real-world application, the Mover and Navigator classes would have a much larger range of methods. The advantage of creating your own classes is that once you have designed and tested the code, it's ready for reuse in any project that you create. Instead of copying and pasting large chunks of code or using include files, you just create an instance of your class and use it like any other part of ActionScript. It makes your code shorter and easier to maintain.

If you find the idea of class-based code daunting, have a look at the differences between Mover and Navigator. You'll see that they're similar in general structure. All classes are similar, and this is the key, because once you build one and have it working, you can make others easily. Although there are all sorts of more advanced substructures and techno-speak to describe classes, the basic class definition isn't that difficult.

Extending a built-in class

When you were reading the previous section, did a little light come on inside your head? Here are a couple of clues:

- When you extend a class, the subclass automatically inherits all the properties and methods of the superclass.
- ActionScript 2.0 has a large number of classes with properties and methods already defined for you.

Not only can you extend your own custom classes, but also you can extend existing ones in exactly the same way. This has potentially massive implications for your code. Let's say you find yourself constantly using a particular function with movie clips and wishing that the Flash development team would add a new method to save you from copying the function into all of your movies. Wish no more—just create a new class that extends the MovieClip class. All instances of your new class will inherit all the properties and methods of a movie clip, *plus* act in the specialist way defined in your class.

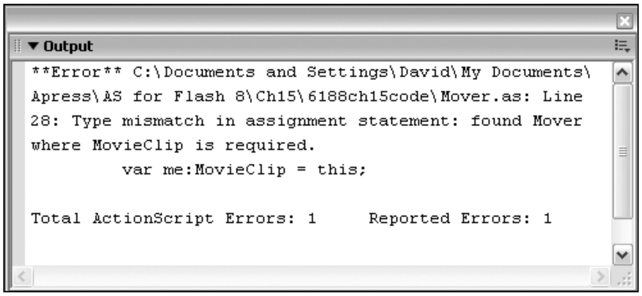
Let's revisit the Mover class to see the implications of this.

Extending the MovieClip class

1. Open Mover.as. If you're using Flash Professional 8, it will open in the ActionScript editing window. If you're using Flash Basic 8, just open the file in a text editor because the changes you're going to make are quite small.
2. Scroll down to the private function moveClip() and amend the first line inside the function to change the data type of me from Object to MovieClip as shown in bold type:

```
private function moveClip():Void {
  var me:MovieClip = this;
  var mc:MovieClip = me._parent;
```

3. In Flash Professional 8, click the Check Syntax button in the ActionScript editing window. If you're using Flash Basic 8, you'll need to save Mover.as and test the FLA that you used in the previous exercises (or use testClass01.fla from the download files). You should see an error message like that shown in the following screenshot.



```
**Error** C:\Documents and Settings\David\My Documents\
Apress\AS for Flash 8\Ch15\6188ch15code\Mover.as: Line
28: Type mismatch in assignment statement: found Mover
where MovieClip is required.
         var me:MovieClip = this;

Total ActionScript Errors: 1     Reported Errors: 1
```

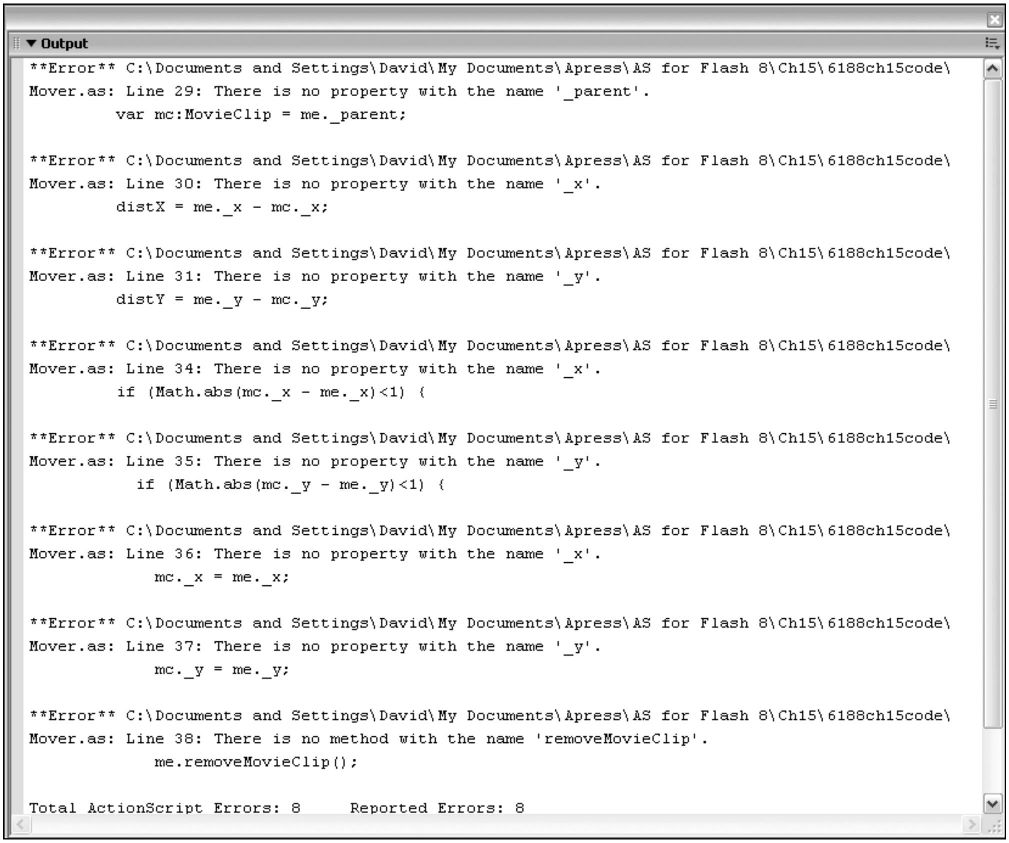
This is where building your own classes can be confusing when you first start doing so. Let's work out what's going on here.

- The `moveClip()` function is used as the `onEnterFrame` event handler for the `private` variable `theClip`, so `this` is a reference to `theClip`.
- In the property definitions at the top of `Mover.as`, `theClip` is declared as being of type `MovieClip`.
- The `moveInertia()` function creates `theClip` as an empty movie clip on `theTarget`, which is your instance of `Mover`.
- Rightly or wrongly, the ActionScript compiler now treats `theClip` as an instance of `Mover` and not of `MovieClip`.

4. Since the compiler expects `me` to be an instance of `Mover`, let's keep it happy. Change the data type like this:

```
var me:Mover = this;
```

5. Check the class again either by using the Check Syntax button (Professional) or by saving `Mover.as` and testing the FLA (Basic). The following screenshot shows what happens.



```
**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 29: There is no property with the name '_parent'.
         var mc:MovieClip = me._parent;

**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 30: There is no property with the name '_x'.
         distX = me._x - mc._x;

**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 31: There is no property with the name '_y'.
         distY = me._y - mc._y;

**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 34: There is no property with the name '_x'.
         if (Math.abs(mc._x - me._x)<1) {

**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 35: There is no property with the name '_y'.
           if (Math.abs(mc._y - me._y)<1) {

**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 36: There is no property with the name '_x'.
             mc._x = me._x;

**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 37: There is no property with the name '_y'.
             mc._y = me._y;

**Error** C:\Documents and Settings\David\My Documents\Apress\AS for Flash 8\Ch15\6188ch15code\
Mover.as: Line 38: There is no method with the name 'removeMovieClip'.
             me.removeMovieClip();

Total ActionScript Errors: 8     Reported Errors: 8
```

Out of the frying pan and into the fire! Not just one error, but eight. But this is where error messages can really come in handy. They tell you that Mover doesn't have the following properties: _parent, _x, and _y. They also tell you that it doesn't have a method called removeMovieClip. Absolutely true—the only properties that Mover has are the four declared at the top of Mover.as.

However, you *do* know a class that has these properties and method: MovieClip. This is a perfect case for extending a built-in class.

6. Amend the very first line of Mover.as to make it extend the MovieClip class like this:

```
class Mover extends MovieClip {
```

7. Now check your class definition again, using the appropriate method for your version of Flash. All the errors have gone. You can now use Mover with any property or method of the MovieClip class. And since Navigator extends Mover, it also inherits all the MovieClip methods and properties, too.

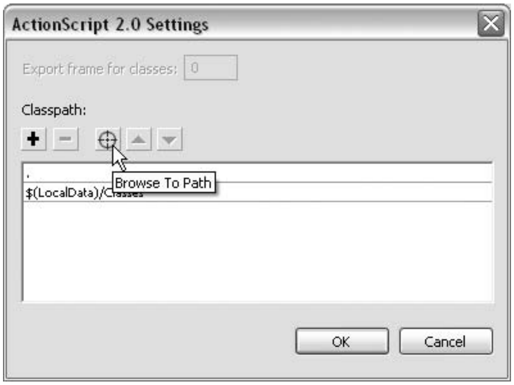
This is a powerful way of extending the functionality of ActionScript, but life becomes complicated if you need to keep copying your class files to the same folder as the FLA you're working on. Fortunately, it's not necessary. You just need to tell Flash where to find them.

Storing classes in a central location

For the purpose of these exercises, we've told you to put the class files in the same folder as your FLA, because that's one of the places that Flash automatically searches when it encounters a class that's not part of core ActionScript. If you start building up a library of your own classes, it makes more sense to keep them in a central location. For Flash to find them, you need to set up a **classpath**. You can do this either globally or for individual FLAs.

Setting up a global classpath

1. Open the Flash Preferences panel (in Windows, it's on the Edit menu, and on a Mac, it's on the Flash menu) and select the ActionScript category.
2. Click the button labeled ActionScript 2.0 Settings at the bottom of the panel.
3. In the dialog box that opens, click the Browse To Path icon as shown, and navigate to the main folder where you keep your classes. This will add the folder to your available classpaths. The screenshot on the right shows a folder on a Windows system called C:\actionscript\classes that has been added as a classpath. Flash will automatically look here—and in any subfolders—for classes at compile time.

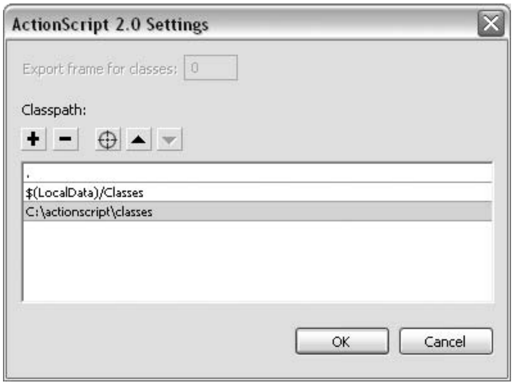


ActionScript 2.0 Settings

Export frame for classes: 0

Classpath:

.
$(LocalData)/Classes



ActionScript 2.0 Settings

Export frame for classes: 0

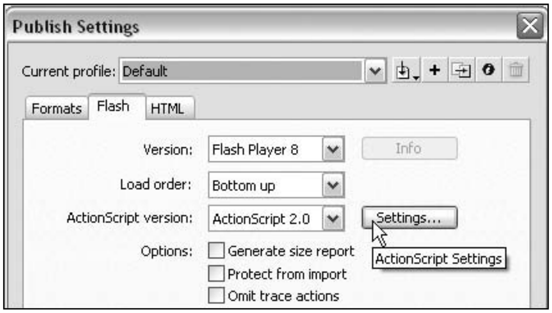
Classpath:

.
$(LocalData)/Classes
C:\actionscript\classes

4. To remove a folder from your classpaths, open the ActionScript 2.0 Settings dialog box as just described, highlight the folder that you want to remove, and click the minus sign button. Your folder and classes will remain intact, but Flash will no longer search that folder or its subfolders automatically at compile time.

Setting up a classpath for an individual FLA

1. Open the FLA that you want to set a classpath for, and select File ➤ Publish Settings.
2. Select the Flash tab and click the Settings button to the right of the drop-down menu labeled ActionScript version. (The button will be grayed out if the drop-down is set to ActionScript 1.0.)



Publish Settings

Current profile: Default

Version: Flash Player 8

Load order: Bottom up

ActionScript version: ActionScript 2.0

Options:
□ Generate size report
□ Protect from import
□ Omit trace actions

3. Follow steps 3 and 4 in the previous section to add or remove a folder and its subfolders from the classpath.

Although we've given you the basic information you need to start creating custom classes of your own, getting the most out of OOP requires a good understanding of object-oriented principles and practices. To help you on your way, Peter Elst and Todd Yard have written Object-Oriented ActionScript for Flash 8 *(friends of ED, ISBN: 1-59059-619-6). It covers all the core aspects of OOP—encapsulation, classes, inheritance, polymorphism, and design patterns—all within an ActionScript context, so it's entirely relevant to your needs as a Flash developer.*

Final thoughts

Well, it's been quite a journey. The first line of code in this book was a simple comment. Remember this?

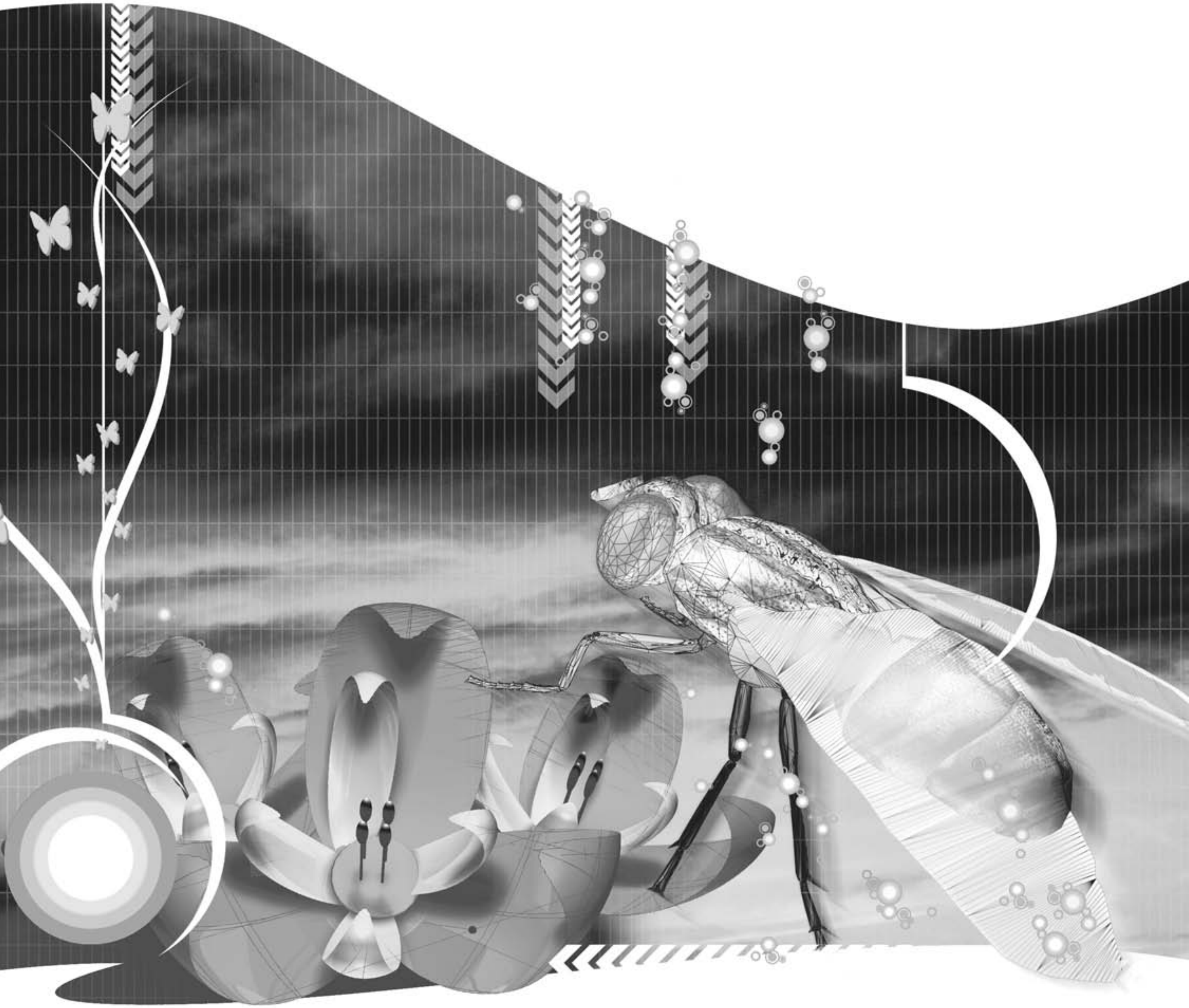
```
// My first line of ActionScript
```

And now look at you! You can work with arrays, functions, objects, and components, and even build your own classes.

You've come a long way, but the journey doesn't end here. In this book, you've gained the fundamental skills and understanding that will allow you to take your ActionScripting onward and upward. The ActionScript language is huge, and it continues to grow, but the techniques you learned in these pages can be applied to all classes. Dip into the ActionScript 2.0 Language Reference from time to time and experiment. We hope that this book has provided you with inspiration and excitement at the massive creative options opened up by efficient and imaginative coding.

Of course, coding is just a tool to do the job. Now that you have the knowledge, we hope you'll start getting a feel for all the things you can do now that you couldn't before. ActionScript is just a way of putting your creative ideas down, mixing them with code, and making something cool.

ActionScript's possibilities are limitless. It's just up to you to `gotoAndPlay`!



INDEX